NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

ROBOT WARS SIMULATION
by

Doreen M. Jones

June, 1997

Thesis Advisor: Gordon Schacher
Co-Advisor: Don Brutzman

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
ROBOT WARS SIMULATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Jones, Doreen M.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT

NPS Combat Systems students learn about robots and autonomous weapons during group design projects in the SE 3015 course sequence is designed to provide combat systems development. The capstone project is the Robot Wars Competition, where pairs of student- designed autonomous robots battle each other. This thesis extends this competition into the arena of simulation and modeling. Our motivation is to further students understanding of the strengths and weaknesses of computer modeling and simulation in combat systems design and testing.

This thesis creates a simulation foundation of the Robot Wars Competition. The simulation has been designed in two main parts, a C++ program that manipulates the Simbots on the playing field and generates data files of their movements, and a 3D graphical visualization that allows the user to see the Simbots in action. The C++ program uses a Simbot class to instantiate two Simbots which are composed of three basic components: base, optics and weapons. The graphics portion uses data files created in the main simulation and displays in 3D animation. The simulation correctly replicates the logical and physical aspects of the robot competition. Future research on the physical aspects of the component parts and the graphics package can be integrated with this foundation.

14. SUBJECT TERMS
simulation, robots, modeling

15. NUMBER OF PAGES 181

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

ROBOT WARS SIMULATION

Doreen M. Jones
Lieutenant, United States Navy
B.S., Tulane University, 1988

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN APPLIED PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
June 1997

ABSTRACT

NPS Combat Systems students learn about robots and autonomous weapons during group design projects in the SE 3015 course sequence. This sequence is designed to provide experience in combat systems development. The capstone project is the Robot Wars Competition, where pairs of student- designed autonomous robots battle each other. This thesis extends this competition into the arena of simulation and modeling. Our motivation is to further students' understanding of the strengths and weaknesses of computer modeling and simulation in combat systems design and testing.

This thesis creates a simulation foundation of the Robot Wars Competition. The simulation has been designed in two main parts, a C++ program that manipulates the Simbots on the playing field and generates data files of their movements, and a 3D graphical visualization that allows the user to see the Simbots in action. The C++ program uses a Simbot class to instantiate two Simbots which are composed of three basic components: base, optics and weapons. The graphics portion uses data files created in the main simulation and displays in 3D animation. The simulation correctly replicates the logical and physical aspects of the robot competition. Future research on the physical aspects of the component parts and the graphics package can be integrated with this foundation.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to express my thanks to those who helped in the completion of this thesis. I would especially like to thank Dr. William B. Maier for helping the Simbot to see properly, and John S. Falby for providing the much needed UNIX commands for the Simbot control.

I. INTRODUCTION

A. OVERVIEW

The SE 3015 Robot Wars Competition is a culmination of four quarters of electronics, control systems and signal analysis course work for masters students in the Combat Systems curriculum. As stated in the NPS course catalog:

> [In the final quarter of the sequence], this course applies the concepts of the Applied Physics I-III sequence to digital data acquisition and control systems. The course covers microprocessor architectures and digital communications using serial, RS-232, parallel, IEEE-488 interfaces, as well as digital and analog interfacing. Two key areas involve the use of small computers for the control of, and data acquisition from peripheral devices. The use of electro-mechanical servo systems for closed-loop feedback control of mechanical devices, such as positioning, pointing and tracking systems. (NPS course catalog, p. 307)

The SE 3015 robot is designed with both standard and student defined parts. All students are provided a base platform, standard gun and beacon. The students are required to build the optical circuit and the fire control circuit for the standard gun using supplied parts and circuit diagrams. Depending on the particular rules for the course, students can add additional weapons or decoys of their own design to the robot. Appendix A contains Robot Wars rules for a recent competition. Following these rules, the students are required to program individual robots with tactics to search, seek and fire upon the other robot on the playing field.

This thesis builds the foundation for extending the SE 3015 Robot Wars competition into the realm of computer simulation and modelling. The physical SE 3015 robot is modeled in software to simulate its physical behavior in response to the external environment and internal control structure. The logic of the SE 3015 robot Dynamic C code (Z-World Engineering,

1995) is preserved as best possible in the simulation environment to accurately recreate the tactics and reactions programmed. This thesis provides a simulation foundation which incorporates only rudimentary aspects of physics in simulating robot motion, optics and weapons systems in order to establish an initial competition simulation version.

B. PURPOSE

The purpose of this thesis is to provide robot code implemented in a simulated competition environment, thereby providing the ability to compare the performance of the simulation to the actual competition. Manipulation of this model can provide insights into the strengths and weaknesses of both simulations and robot tactics. The simulation foundation was chosen to touch on all aspects of the Robot Wars Competition so as to create a working model for use in future development.

C. THESIS ORGANIZATION

This thesis consists of overall descriptions of the SE 3015 robot, its control structure, the simulation robot (Simbot) and the major components of the robot. This thesis covers the basic design of the robot wars computer simulation, the setting up the initial robot class and the graphics display environment. Initial models of the base, optics, and weapons have been created for use in the simulation.

Chapter II describes the problem being studied in detail. Chapters III and IV provide a general description of SE 3015 robot and its control structure, detailing the parameters to be used and the functions mapped to the simulation. Chapter V covers the simulation environment,

simulation robot and their control structure. The SE 3015 project robot (Figure 1.1) consists of three main hardware portions: Chapter VI for the robot base, Chapter VII for the optics, and Chapter VIII for the weapons. Chapter IX discusses the targeting and tactics used in the Robot Wars Competition.

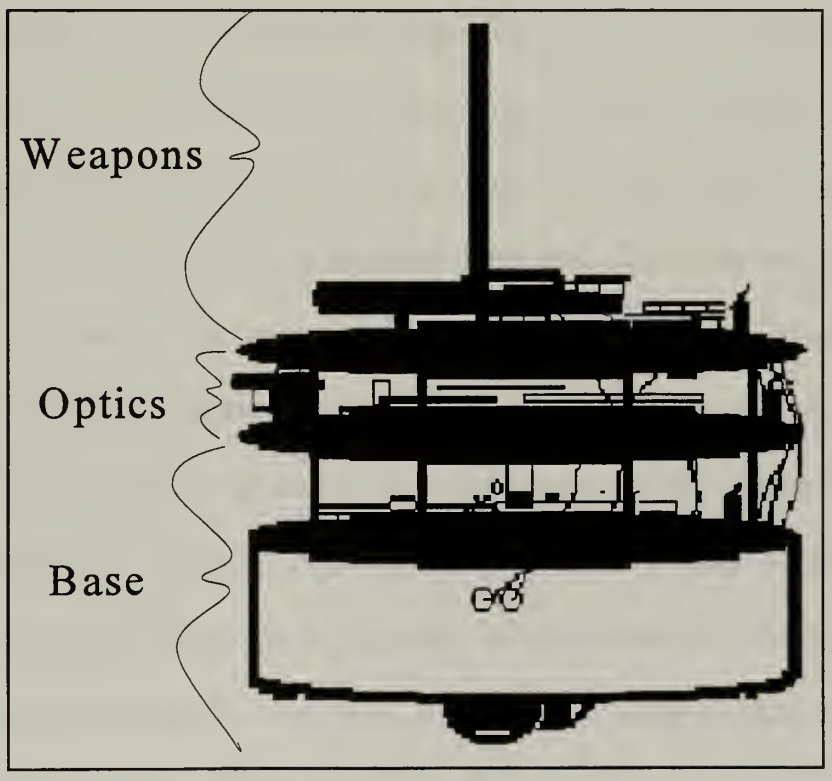
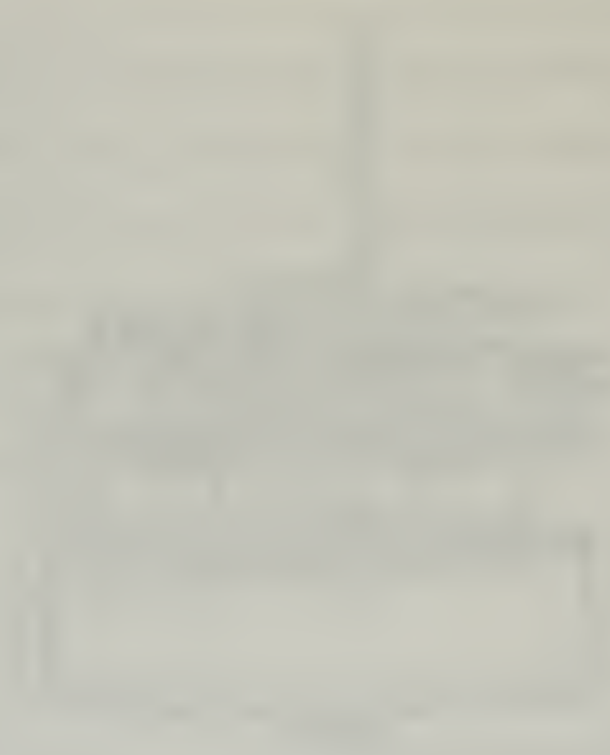


Figure 1.1 SE robot side view



II. DESCRIPTION OF THE PROBLEM

The scope of this thesis is to create a "foundation" model of the "Robot Wars" robot that can interact with its opponent on a simulated playing field. The foundation is designed to allow future student research to concentrate on improved simulation of one specific area while having the capability to simulate the competition for testing and analysis. This work also provides a visualization of the Robot Wars Competition to observe the interaction of the robots on the playing field, thereby permitting analysis and improvement of tactical software.

In order to simulate the Robot Wars Competition, a computer model of the physical robot is created to interact on the playing field with opponents. The robots used in the SE 3015 Robot Wars Competition are built in three parts: the base, the optics and the weapons systems. Similar in construction, the bases are supplied to the student. The base controls the robot motion and houses it's computer. The optics allow the robot to see its opponent and the weapons system is used to attack the opponent. Optics and weapons are constructed by the student using supplied circuit diagrams and parts, but each student can modify them to fit their requirements. To constrain the development of the simulation, the differences between robots are limited to the optical circuit and the weapons systems. The robot base units are considered identical and assumed in perfect working order.

The SE 3015 robots are programmed in Dynamic C (Z-World Engineering, 1995) and function autonomously. They interact with each other only if they detect their opponents beacon with the optical sensors or bump into each other on the playing field. They also interact with the playing field edges. The simulation is required to preserve the autonomous interaction

of the robots, and also maintain the logic and control systems flow. The Dynamic C code of the robot is integrated into the Simbot's C++ simulation code. The tactics and logic are preserved to replicate the actions and reactions expected of the original SE 3015 robots based on the Dynamic C code.

The Robot Wars simulation foundation developed here incorporates a C++ class structure for the robot, a simulation control program to manipulate the robot players on the field, and an animation program to provide a 3-dimensional (3D) display of the simulation.

III. THE SE 3015 ROBOT

To create the foundation simulated robot, Simbot, the SE 3015 robot must be defined and broken into its major physical components and parameters, its code interpreted and mapped to the simulation, and its operations analyzed. The incorporated components are the robot base or movement, optics or vision and weapons. For the baseline simulation, many of the non-critical components have been omitted and the parameters ignored to simplify the initial simulation. This section describes which components and parameters were the incorporated or ignored to create the ideal Simbot. The standard gun breakdown is shown as well as the parameters which are mapped to the simulation and a typical playing field.

A. ROBOT COMPONENTS

The following is a breakdown of the basic SE 3015 robot, including components and physical attributes that affect its performance. Although many of these items have been ignored they are listed for completeness and as a guide for future work.

The attributes which are in **bold** are considered to be the most critical to the simulation. These are attributes for which measurements and assumptions have been made which directly alter the Simbot's behavior. An example of such assumptions is to take the conditions of the attribute to be ideal, such as a uniform weight distribution for the robot. By assuming the robot to be perfectly balanced, the effects of dragging the stoppers along the floor and subsequent friction can be ignored. This assumption were made to create the Simbot and show its movement as directed by the code. In actuality, an off-centered robot will move slower and not properly follow directions from the code. This could also effect the firing angle of the weapon.

The specific components and attributes italicized in the list have been ignored in the simulation. The complexity of the measurements and the lack of robot development standardization contribute to the difficulties in simulating some of these items. For example, the signal levels obtained from the photo diodes are a function of the nose/eye geometry, circuit design, and the batteries used.

Some attributes do not cause significant variation to the simulation if modeled. For example, the energy level of batteries used in the robot can affect the performance if they are not stable and within adequate charge levels. The robot motor may function perfectly while the battery is fully charged, but fail to function (i.e., move the robot) if the battery voltage level falls too low. There is no detected intermediate situation where the robot motor will function but not at peak efficiency. This is in contrast to the eye circuit which, when the batteries begin to get low, will start sending erroneous reading along with the good readings before complete failure. However, if the student keeps the robot main battery charged as instructed and uses fresh batteries for all other cases, the robot will perform the same on each use. The effects of the batteries only become important after extended usage. In this simulation, none of the performance characteristics are considered battery dependent and therefore all batteries have been ignored. Similar lines of reasoning have been used in deciding which of the many parameters, listed in Tables 3.1 through 3.4, are modeled in the simulation.

The primary functions of the SE 3015 are movement, targeting and firing. The movement (as described in Chapter VI) is a function of the motor and wheels as directed by the robot code commands. The targeting determined by the signal level of the opponent beacon as

1. physical robot
   - a. **weight**
   - b. **height**
   - c. **diameter**
   - d. *rubber stoppers (balance)*
     - i. *placement*
     - ii. *height*
2. motor - drive train
   - a. **angular velocity**
   - b. *angular acceleration*
   - c. *friction*
   - d. *torque of drive shaft*
3. wheels
   - a. **radius**
   - b. *width*
   - c. **placement**
   - d. **linear velocity**
   - e. *linear acceleration*
   - f. *friction*
4. power supply (main battery)
   - a. *voltage level*
   - b. *stability*
   - c. *battery life*

5. edge detectors
   - a. **placement on robot**
     - i. fore edge detector
     - ii. aft edge detector
   - b. *tolerance*
   - c. detection method
     - i. *optical*
     - ii. EMF
6. bumpers
   - a. **placement on robot**
     - i. fore bumper
     - ii. aft bumper
   - b. *obstructions*
7. computer
   - a. *baud rate*
   - b. program type
     - i. **while loops**
     - ii. *costatements*

Table 3.1 Robot Body

1. photo detectors
    a. *height on robot*
    b. *position on robot*
        i. *angle*
    c. **geometry of detectors**
        i. **nose length**
        ii. **detector angle relative to centerline**
2. circuitry
    a. *number of circuits*
    b. **cross talk/interference**
    c. gain
    d. bandwidth
3. battery
    a. *voltage level*
    b. **voltage stability**
    c. *battery life*

Table 3.2 Robot Optics

1. *circuit board*
2. *beacon body*
    a. *beacon alignment*
3. *placement on robot*
    a. *height*
    b. **obstructions**
4. *batteries*

Table 3.3 Beacon

1. standard gun
    a. *barrel*
        i. **length**
        ii. *radius*
        iii. *friction*
        iv. *angle*
    b. *magazine*
        i. *length*
        ii. *radius*
        iii. *friction*
    c. motor
        i. *speed*
            (1) *power supply*
        ii. *wheel radius*
            (1) *friction*
            (2) *o-ring type*
    d. firing solenoid
            (1) *duration of signal*
            (2) *length of pin*
    e. projectile velocity
        i. *internal velocity of barrel*
        ii. **exit velocity**
    f. **dispersion**
2. ammunition
    a. size
        i. **uniform/spherical**
        ii. other shapes
    b. weight
    c. density
    d. shape
    e. **drag**

Table 3.4 Robot Weapon

detected by the optic circuit which is then evaluated by the robot programmed tactics (as described in Chapter IX) to determine the proper reaction or action. The weapons (as described in Chapter VIII) are fired based on a targeting solution determined by the robot code.

B. ROBOT PARAMETERS

The Simbots have been set with default parameters chosen to simulate typical behavior. The specific incorporated parameters, their corresponding variable names in the simulation, and default values are listed in Table 3.5. These are the physical parameters that have been represented in the simulation. These parameters are modifiable within the class by changing the Simbot Class object parameters.

Some of the variables will have more than one name to prevent confusion when switching between the simulation main program and the robot class. Where applicable, measurements are given first in English units followed by the simulation units. The scale is approximately 1 foot to 100 grid units. Simulation units were used to conform to the graphics package used to provide the simulation animation. This scale converts the playing field measurements of 16' x 16' to 1600 x 1600 grid units and the robot diameter of 12.75" to 100 grid units.

The eye widths and nose length are set at a ratio of 1:1.5. This ratio makes the limiting angle 33°. The limiting angle is critical to left and right discrimination and the robot's ability to determine the relative location of its opponent on the playing field; see Chapter V, Section D for further discussion. The wheel radius and separation distance are scaled at 1:4 for convenience of assessing the rotation angles and movements. The default angular velocities of the wheels arerepresented by a range of velocities -15 to 15 as sent by the motor control string, described in

Chapter V, Section B. The true change in velocities is nevertheless non linear from one to the next (i.e., speed 4 is not four times faster than speed 1). The nonlinearity of the wheel speed changes is evident when the robot cycles through the wheel speeds. Although wheel speed changes are not calibrated, they are not a major factor at this point in the simulation development.

parameter	simbot Class variable name	default measurement
playing field size	floor, floor size	16 ft (1600 units)
robot radius	body	1 ft (100 units)
max eye signal level	MAXEYE	2040
background eye signal level	background	16
eye nose length	height	1.5
right eye width	R_width	1
left eye width	L_width	1
wheel radius	radius	0.25
wheel separation	separate	1.0
wheel angular velocity	--------	---------
left wheel	w1	-15 to 15
right wheel	w2	-15 to 15

Table 3.5 Robot Parameter - simulation variable mapping with default values

C. WEAPONS MODEL AND PARAMETERS

The standard gun is the only weapon modeled for the simulation foundation. The standard gun and bullets are mapped to matrices that contain both their parameters and

information concerning their current status. Currently, the simulation includes only the exit velocity of the standard gun and the magazine size. The bullets are held in a single magazine (matrix) of m x 8, where m is the total number of bullets available. The weapons matrix is n x 4 where n indicates the number of weapons types available. A breakdown of the bulles matrix is contained in Table 3.6, and the weapons matrix appears in Table 3.7.

row element	parameter description	default value
bullet[x][0]	status flag 0 - not fired 1 - active 2 - spent	0 - not fired
bullet[x][1]	launch angle	Simbot gun angle set at time of launch
bullet[x][2]	exit velocity of gun launched from	
bullet[x][3]	x position	Simbot x position set at time of launch
bullet[x][4]	y position	gun[x][3] value set at time of launch
bullet[x][5]	z position	Simbot z position set at time of launch
bullet[x][6]	hit or miss indicator	
bullet[x][7]	gun launched from	

Table 3.6 Bullet matrix row elements

row element	parameter description	default value
gun[x][0]	magazine size	15 (constant value)
gun[x][1]	rounds available	15
gun[x][2]	exit velocity	105 ft/sec (10500 units/time)
gun[x][3]	launch height	1.5 ft (150 units)

Table 3.7 weapons matrix row elements

D. PLAYING FIELD

The SE 3015 Robot Wars competitions is held on a 16' x 16' playing field which is ringed by a wire to facilitate edge detection as shown in Figure 3.1. The wire used to line the edge emits an EMF signal of 22 kHz. The SE 3015 robots are equipped with edge detectors the robot base, which when detecting the signal will instruct the robot to move away from the edge; see Chapter V, Section C.3 for additional details.

E. SUMMARY

The SE 3015 is easily decomposed into the base, optics and weapons. The basic parameters of the SE 3015 robot, weapons and projectiles of the weapon have been mapped to the simulation through a robot class and matrices.

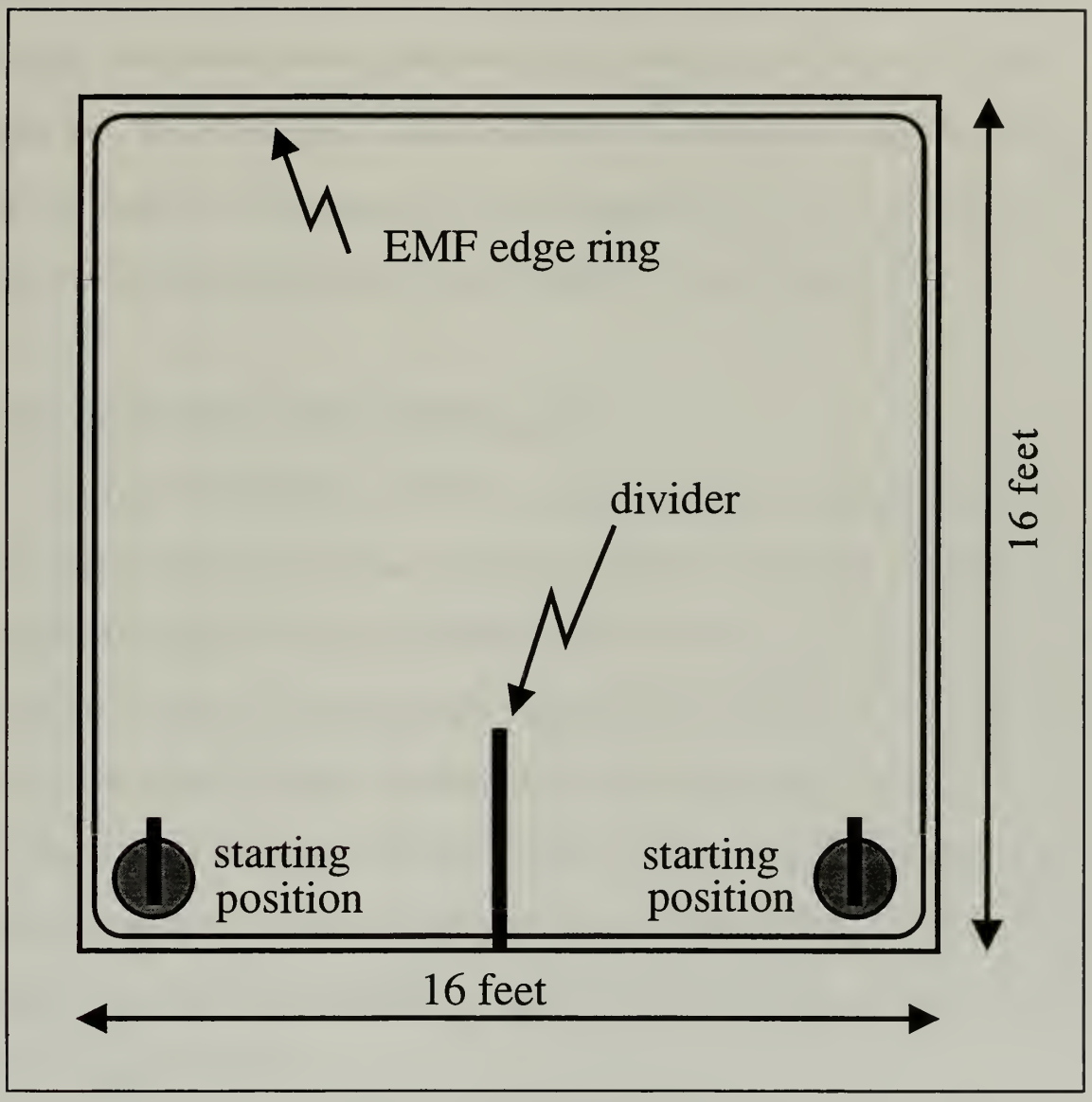


Figure 3.1 Robot Wars playing field

IV. ROBOT CONTROL SYSTEM

The robot actions and reactions to its opponent and the playing field are controlled by its computer code. Programmed in Dynamic C (Zworld Engineering, 1995), the code controls movements, interpretation of the signal levels received from the eyes, tactics, reactions and weapons fire. One of the primary goals of this thesis is to translate the Dynamic C code of the robot to the Simbot C++ code, preserving the tactics and actions of the original robot code. This section describes the translation process and its integration into the simulation.

A. TINY GIANT MINIATURE CONTROLLER

The Tiny Giant Miniature controller is a compact miniature controller produced by Zworld Engineering (Zworld Engineering, 1993). It consists of a Z180 micro processor with a 9.216 MHZ clock speed. 256K bytes of EEPROM and 512K bytes of battery packed static RAM are available to the student. The Tiny Giant is equipped with a 4-channel analog-to-digital (A/D) converter with configurable input amplifiers which can be accessed with a Wago connector. The Tiny Giant is programmed using a Dynamic C interface board. The program is transferred from an IBM-compatible PC through a Zworld Z485 communications processor card via an RS232 serial port. The Tiny Giant is capable of operating in a stand-alone capacity once a program has been downloaded into RAM.

B. DYNAMIC C CODE

The robot code is written in Dynamic C, a version of C tailored for use with the Tiny Giant. An IBM compatible PC is required for programming the Dynamic C code. Dynamic C

allows for the quick downloading via the serial port and testing. While connected to the computer, data from the SE 3015 robot can be printed to the computer monitor. This capability is most often used to check the signal levels obtain from the robot eyes.

The Dynamic C code for the robot can be written with a variety of different styles to include while loops for sequential processing, and costatements for parallel processing. The majority of students prefer to use while loops, which will have the SE 3015 perform its actions sequentially, looping through portions of the program until a desired condition is reached (e.g., the opponent is within firing range). The costatement approach allows for several actions to be running at one time (i.e., the robot can look at the same time it is moving.) The simulation currently only allows for while-loop sequential processing. Appendix B, provides a sample robot code written with while loops.

C. ROBOT OPERATION

The robot is controlled by the student-defined program that is cross-compiled in Dynamic C and downloaded into the Tiny Giant controller. Each program written requires parallel input/output data ports A and B (PIODA/B) to initialize the output ports. The PIODA ports are used to receive the information from the optical sensors in the form of an integer, currently between 16 and 4080 (previous maximum value was 2040). The value 16 corresponds to background noise and 4080 corresponds to saturation when the other beacon is immediately in front of the photodetector. The PIODB ports are used to send signals into the robot to turn on the gun motors and fire the gun.

Each robot code requires two functions: the platform function, and the interrupt handler. The platform function sends the motor control string (an eight character string indicating motor direction and speed) to the robot motor to control the robot movement. The SE 3015 robot has two powered wheels each of the two wheels has its own motor. The motor control string provides the necessary commands for each motor. Table 5.1 provides a description of the motor control strings.

The SE 3015 robot code has an initial one-time set up to initialize variables and input/output ports that is separate from the primary search and shoot routine that drives the robot. This initial portion may also include a small program to move the robot from the starting position, usually a corner of the playing field, into a more advantageous point generally in the center of the playing field to begin its search. Then the robot typically enters a search routine, (generally within a while loop) to search for, locate, move to and fire upon the opposing robot. This portion of the program incorporates a sense-decide-act logic loop. The sensing is performed by getting readings from the right and left eyes. Decisions are then made based on the combined value and the individual values of the eyes as to where the opposing robot is on the playing field. The decision also includes the determination as to whether to move closer to the other robot or fire upon it. The "act" result is either a motor control string sent to the platform function to move the robot or a fire signal sent to the weapon.

At any time (although most often during a movement phase) an interrupt can be sent to the program from either the edge detectors or the bumpers. This interrupt will halt the main program, execute the interrupt program based on the type of interrupt, then return to the main program. In most cases, the main program will resume where it was interrupted, unfortunately,

there is a possibility that the interrupt can cause the program to "forget" where it was when interrupted. When this occurs, the program may restart at the beginning, return to the top of the current loop or reexecute a command that was previously completed. On rare occasions, robots have "gone to sleep" for a few moments before continuing. Figure 4.1 shows a simple logic flow of an SE 3015 robot.

D. SE 3015 FUNCTION MAPPING

The simulation attempts to preserve the robot code as best possible by using the same (or similar) functions and I/O calls as the robot uses. This section will show how the function and I/O calls are mapped to the simulation. The physical parameters are described in Chapter III, Sections B and C. Some components of the robot require only a single function to simulate their behavior while others require multiple functions.

Table 4.1 shows the robot I/O and function calls and variables that need to be directly mapped to the simulation from the SE 3015 robot code. Appendix C provides descriptions of the robot class and the related functions. This is Dynamic C code that is student programmed and then downloaded into the actual robot for the competition. The Simbot code is translated from the Dynamic C code for use in the simulation main program; Appendix D provides specific details on converting the robot code.

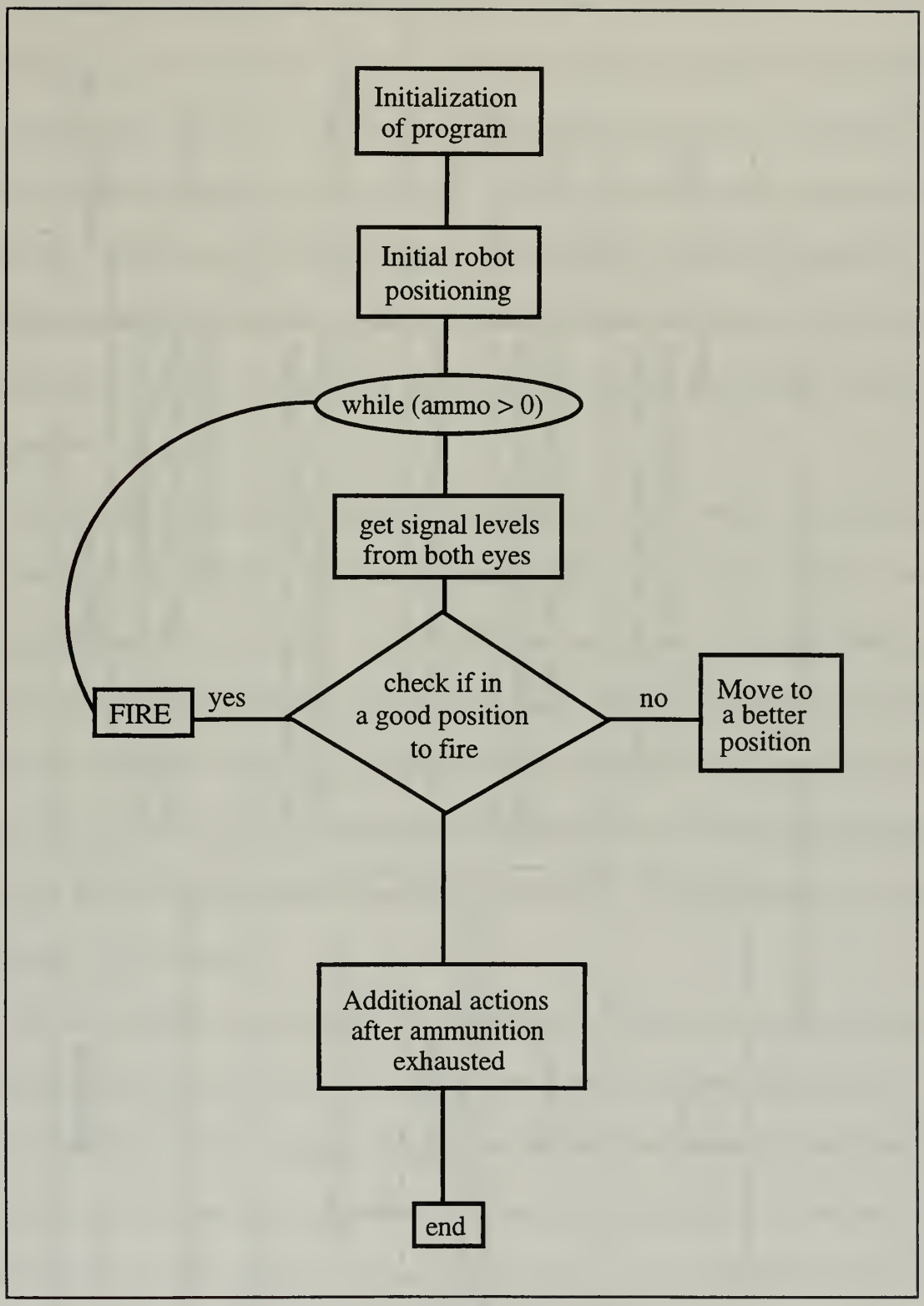


Figure 4.1 Sample SE 3015 code logic flow

SE 3015 Robot	description	SIMBOT
ad_rd(int n)	I/O call which returns an value from the eyes calls the class function optics to calculate the signal levels of the eyes.	botA.ad_rd(int n, SimbotClass & botB)
outport(PIODB, 0x0a)	sends an output signal to a component of the robot such as the weapon, indicating a signal to fire	botA.outport(PIODB, 0x0a)
platform( )	sends the motor control string to the motor	platformA( ) botA.platform(SimbotClass & botB)
buf	character string to hold the motor control string sent to the platform function above	botA.buf
count	counter to determine the length of time between motor control string sends to the motor	Acount

Table 4.1 Code mapping table for SE 3015 code to simbot code

The significant different between the Dynamic C code and the Simbot C++ code lies in the specification of the function or I/O call to a Simbot. The robot computer is internal to the robot and drives the robot based on the Dynamic C code and external input. The SE 3015 robot is only aware of its opponent through its sensors. The simulation computer is external to the Simbots and is controlling both at the same time. The simulation computer is aware of the parameters of both robots at all times. However, the simulation needs to know which Simbot is calling a function and which Simbot is the object of the function call (i.e., either Simbot A is looking at Simbot B or vice versa).

The simulation must link the function or variable to the Simbot object by use of the **botA.** prefix attached to the robot class variables and function calls. The exception to this is the function **platform** which is a function inside the main and used to control the simulation flow. Each robot must have its own platform function as it has its own robot simulation code function (as described in Chapter V Section B) provides a detailed description. The function platform is used by both the SE 3015 code and the Simbot code to convey the motor control string to the motor. The Simbot function **platform** then calls a specific class function **platform** which then calculates the robot movements.

The robot actions and components are mapped on the simulation as shown in Table 4.2. The Simbot functions shown are all class functions and accessible only through the class functions themselves, with the exception of the functions **platform**, **outport** and **ad_rd8**. The **platform** function is called from the **platform** functions in the simulation main program. The **outport** function is used to interface the parallel I/O data ports and the **ad_rd8** function is used to obtain the simbot optics signal level. Each of the Simbots instantiated by the main program

SE 3015 Robot	description	SIMBOT
motor control	breaks the motor control string into the specific information for each wheel	getVelocity(float w1, float w2)
wheel movement	calculates the movement of the simbot over the next time step	position( ) platform(SimbotClass &)
edge detection	checks to see if the simbot is at the edge of the playing field, determines if it is the fore or aft edge and then responds accordingly	edge_detect(float w1, float w2)
bumpers	checks to see if the robot has hit and obstacle on the playing field and then moves always from the obstacle encountered.	bumpers(SimbotClass &, float w1, float w2)
photo diodes	the eyes of the robot. The optics function computes the Simbot A's relative position to Simbot B and then returns a signal level to the ad_rd8( ) interface	optics( ) relative_angle(SimbotClass& bot2); range(SimbotClass& bot2)
weapons fire	sets the launch conditions and position for the projectiles	gunfire( )
projectiles	computes the flight path and scores the projectile	projectiles( )

Table 4.2 Component and action mapping for SE 3015 hardware and performance attributes to the simulation robot class functions

has a set of these functions which can be tailored to the individual Simbot through the initial instantiation process. The list of incorporated parameters used in these functions is found in Table 3.5.

The additional functions or actions that affect the Robot Wars competition are the scorers and referees, shown in Table 4.3. Their actions are part of the robot class and perform similar functions but are not a simulation of the specific behavior performed. The **auto_ref** function works to keep the Simbot from leaving the playing field or becoming stuck in the bumper routine. The scoring of the Simbots is strictly based on the projectile calculations. Both of these functions are missing the human factor and the randomness of their associated behavior.

E. SUMMARY

The SE 3015 robot is controlled by the Tiny Giant Micro controller which acts as the robots computer. It uses several functions to receive information from its eyes and sends information to the robot motor and the weapons. These function are mapped onto the simulation robot class functions along with the human referee and scorers.

SE 3015 Robot	description	**Simbot**
referee	will move Simbots stuck in the edge routine or bumper routine or combination there of	auto_ref()
scorers	assesses whether or not the projectile has hit the Simbot.	projectiles()

Table 4.3 Additional Robot Wars Competition functions mapped to the simulation robot class functions.

V. SIMULATION

The Robot Wars computer simulation is written in a combination of five computer languages and is composed of five parts. The simulation flow must allow the Simbots to act nearly simultaneously with each other in order to preserve the integrity of the original simulation. The simulation manipulates two Simbot class objects which represent the opponents and generates two data files containing their position information at each time step. The graphical program creates a computer generated animation display from these data files to show the robot interactions.

A. OVERVIEW

The simulation consists of the main program, the robot simulation code, the robot class, the robot data files, and the animation display. The main program is written in a combination of C++ and UNIX C and is the controller for the simulation. The robot simulation code was originally written entirely in Dynamic C for SE 3015 Robot Wars and converted as necessary to C++ for the simulation and is contained in the main program. The robot class contains the computer simulation of the robot components and is written in C++. The data files are ASCII text files created by the robot simulation and used to display the animation in the animation program. The animation program is written in a combination of Open GL (Nieder, 1993), XMOTIF (McMinds, 1993), and C++ and displays a simple animation of the robot wars using primitive forms. Table 5.1 provides a breakdown of these parts which will be discussed in this chapter.

program file	description	number of files/objects generated.
main	The main file contains the two sets of the Simbot code, platform function and interrupt handler for each of the Simbot. Instantiates two Simbot objects and two processes. The main controls the simulation flow through process handling	1
SimbotCode	a function of the main program containing the converted Dynamic C code for use to control each of Simbot actions/reactions.	0
Simbot Class	The Simbot class contains the functions simulating the robots hardware and software	2 objects instantiated
data file	The date files contain the position information of the Simbots generated by the main program for use in the graphics animation program.	2 files, (one for each Simbot)
graphics program	reads the data files and graphical display the robots and movements through computer generated animation.	1 (plus related supporting files)

Table 5.1 Program file descriptions.

In order to properly simulate the SE 3015 robot war, the program must be able to interact with both robots and have them move or act simultaneously or nearly simultaneously. Near simultaneous movement is defined as one robot moving a small incremental step, then the other a similar sized step, alternately repeating the sequence. These motions must replicate the motions as directed by the students robot code used in SE 3015 in order to maintain the integrity of the simulation. Each Simbot must be able to obtain the sensor information, optical or contact, on the other Simbot in response to the other Simbot's movement. After a Simbot fires on the other robot, the program must be able to determine if the projectile impacts the other Simbot which may have moved from its position at time of fire. Additionally, after each step, a Simbot must

be able to determine if it has made physical contact with the other Simbot, hit an edge or has acquired the other Simbot with its optical sensors.

In order to obtain this near-simultaneous movement while preserving the students' original code, UNIX C commands were used to create a separate process for each of the Simbots. Each Simbot code function is a process of the main program. The processes control the Simbots movement on the playing field and through its own Simbot code. The main program must be able to track both Simbots place within their respective codes, and when exiting one Simbot's code must be able to return to the other code where it had previously left. If the main program does not know where it last left a particular Simbot code, it will start at the beginning of the function each time it returns. The simulation main code is found in Appendix D.

B. SIMULATION PROGRAM COMPONENTS

1. Main Program

The main program uses UNIX C to create two separate processes, one for each of the Simbots, and a sharing routine. The main program controls the sharing of the two Simbot processes and the master clock. The Simbot movements are controlled in the Simbot simulation code and the Simbot class. The main program instantiates two Simbot class objects that will simulate the actual robots. The program uses the UNIX C commands **blocproc** and **unblocproc** to hand off control to the two Simbots. The program terminates when the master clock reaches the simulation-end time, set at the start of the program.

The command **blocproc** blocks the current process and allows the computer to return to the main program to get the next instruction. The command **unblocproc** allows the next process

to begin. Only one process may be active at a time. The processes are assigned their own process identification that the control uses to identify which is active and which is asleep.

The master clock count, although not a one-for-one ratio with seconds, can be thought of as approximately the number of seconds to run the simulation. A clock count equals one movement of the robot on the playing field. All logic decisions, sensor readings, and weapon fires will be considered as instantaneous events in this simulation. At the completion of the simulation, the main program displays the Simbot scores.

2. Robot Simulation Code

The robot simulation code contains the student's Dynamic C code converted to C++, and the modified **platform** function and interrupt routine. One set of the functions is created for each of the Simbots and placed in the main program. Each Simbot has three functions within the main program. A minimal amount of modification is done to the Dynamic C code by the student to conform to C++ for use in the simulation. Conversion details are provided in Appendix E.

The **platform** call in the robot code requires further explanation about its conversion from Dynamic C to C++. In the Dynamic C code, the **platform** function is called and then followed by an empty "for loop" to delay any further platform calls until the robot has had a chance to handle the first platform call. For example, the robot's microprocessor requires a certain amount of time to properly handle the motor control string sent to it via the robot code. If motor control strings are sent too close together, the robot will not be able to process the string properly, resulting in a timing error that may hang up the robot microprocessor. This, however,

does not pose a problem for the Simbot, where new motor control strings can be sent immediately after each other.

With the actual robot, the motor control string is sent by the platform to the motor. The robot then moves according to directions indicated by the motor control string for the duration of the "empty for loop" and until a new motor control string is sent to the motor. This is not the case with the simulation. Since the simulation is written in C++, most compilers will optimize the empty for loop and will not run it as a delay. For example, the code line below

for (count = 1; count < 1000; count++);

is a null statement and will become

count = 1000;

after being compiled. Initial tests of the Dynamic C program, as written for SE 3015, in the simulation had the Simbot making one incremental move based on the first motor control string, ignoring the empty for loop and proceeding to the next motor control string and platform call. So, instead of going forward for a count of 100, the Simbot went forward for a count of only 1. Changing the order of the "for loop" and the platform call in the simulation, Figure 5.1, forces the simulation to make successive platform calls based on the old motor control string, not allowing the simulation to proceed to the next motor control string until the delay has been simulated. For example, the actual robot would send one motor control string, act on it and then counts to 100 while moving in accordance to the control string before continuing with its code. In the simulation the robot is sent the motor control string once but calls the platform function 100 times, making one incremental move per function call.

code fragment for Dynamic C robot code

```
strcpy(buf, "ffgrffgr");                    (1)
platform();                                 (2)
for(count = 1; count < 1000; count++);      (3)
```

modified code fragment for use in simulation

```
strcpy(bot.buf, "ffgrffgr")                 (1)
for (bot.count = 1; bot.count < 1000; count++ )  (3)
   platform();                              (2)
```

Figure 5.1 Platform function call sequence translated into C++

The simulation platform function calls a specific Simbot class function assigned to a Simbot. It is within these class functions that the actual movements and the interaction between Simbots are calculated. In most cases the robot code is modified by added a **botA.** or **botB.** before all I/O commands. **POIDA(0x12)** becomes **botA.PIODA(0x12)** in order to access the correct Simbot class functions. The **platform** function call is changed to either **platformA** or **platformB** depending upon the Simbot. The **platform** function is modified from the original platform function used by the student to call the class platform function and to handles the process control. The platform function writes the position information to the data file.

3. Simbot Class Functions

The Simbot class is used to simulate the robot hardware. It is composed of a set of generic movement, projectile, and optical sensor functions. These functions calculate the actual movements, sensor reading and projectile trajectories. The functions can be modified to better

simulate the individual robots by hard coding the changes into the simulation by use of virtual function designations. Each function is discussed in Appendix C.

The interrupt routine is overwritten as a class function called **edge_detect** and is linked to the robot class. The response to the forward or aft interrupt is not altered from the students original logic only how it determines the interrupt. This new edge detection routine is called by the Simbot class **platform** function as part of the new position calculation. The bumpers are handled within the Simbot class with a universal routine to be used by all instantiations of the Simbot class.

4. Data Files

The Robot Wars simulation generates one data file for each Simbot. The data file contains, the time count, the Simbot color, the x and z positions of the Simbot on the playing field and a fire flag designating whether or not the Simbot has fired a projectile. The fire flag is either a 0 for no projectiles in the air or a 1 for projectiles in the air, this is used to allow the graphics program to change the Simbot color when it has fired a projectile.

C. 3D GRAPHICS PROGRAM

This program is an XMOTIF/OpenGL program written to display the Simbots on the playing field. The program uses primitives to show the playing field, Simbots, and projectiles. The program reads in the information from the data files to draw the primitive shapes corresponding to the Simbot movements determined by the data. The simulation allows the display of the entire Robot Wars or selected portions by indicating the start and end times. The

simulation allows for different viewing angles of the robot wars in progress. A single light source is used with no texture mapping to allow for near real time simulation speeds. The simulation can be sped up or slowed by controlling the floor pattern. The single square pattern on the floor will run the program at its fastest speed while the checkered floor will result in slow motion.

The animation portion was kept separate from the rest of the simulation to better allow for display improvements. Using data files that are not platform specific, the animation display can be written in any language that will support reading ASCII text files. In the future the animation portion may be modified to allow for display over the internet.

D. ROBOT WARS SIMULATION FLOW

The flow of the simulation execution is shown in Figure 5.2. This flow simulates the constant interaction between the two Simbots on the playing field. Unlike the actual SE 3015 Robot Wars, each Simbot will know the position of itself and the other Simbot on the playing field. The following details the interaction between the main program, the students robot code, the robot class and the data file creation.

1. Initialization

The program starts with the instantiation of two robots, **botA** and **botB** and two processes. Each of the Simbot's code is a designated processes, assigning **processA** to **simCodeA()** and **processB** to **simCodeB()**. The simCode is the function while, the process is

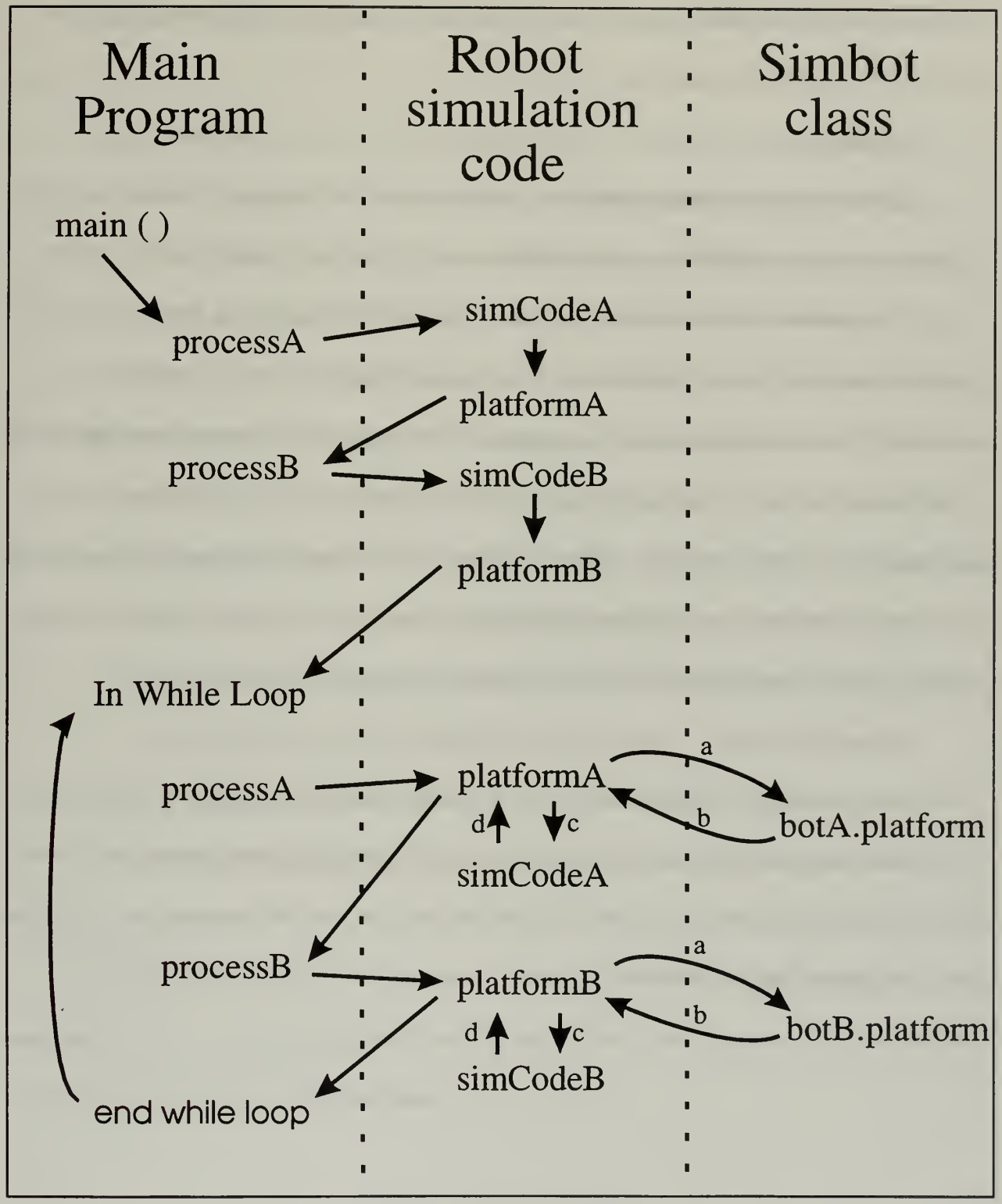


Figure 5.2 Simulation flow

the actual performance of the Simbot code. The master clock count is designated by the user and will determine the duration of the simulation.

2. Start Up

Simulation start up requires that each process start at the beginning of its **simCode** and exit when it reaches its first Simbot class **platform** call, but before it makes it first movement.

The **processA** will initiate the beginning of the **simCodeA** and work through the code until it reaches a platform call, **platformA( )**, the process then blocks itself and unblocks **processB**. This occurs before entering its specific Simbot's platform function, **botA.platform()** to calculate its first move. It is important to note that **processA** exits from **platformA( )** vice **simCodeA** where it first entered to facilitate the hand off to **ProcessB**. **ProcessB** will repeat the same steps as **processA**, exiting through **platformB( )** and returning to the main program after blocking itself and unblocking **processA**. The main program then starts the while loop.

3.The While Loop

The while loop will repeat the hand-off between processA and processB for the duration of the count designated by the master clock. The simulation will then step through the Simbot code, determining the Simbot's actions in accordance with the simCode and reactions to the other robot and playing field boundaries.

ProcessA( ) starts by returning to the **platformA( )** function and then follows the steps below.

step a: **platformA( )** and enters **botA.platform( )** to calculate its incremental step. **botA.platform( )** will make additional function calls to other class functions. The class functions will convert the motor control string into angular velocities, determine if the Simbot has hit an edge or the other Simbot and determine the necessary reaction, and calculate its new position before returning to **platformA()**.

step b: The process returns from **botA.platform( )** to **platformA()** and all new position data is written to the related data file. The process then returns to **simCodeA()**.

step c: The process returns to the **simCodeA()** to just after it had made its last platform call. The process then works through the simCode until it reaches its next platform call. Only movement instructions are considered a time step, all other decisions or I/O calls are considered instantaneous.

step d: The platform function call then returns the process back to **platformA**. At this point, **processA** blocks itself and unblocks **processB**, returning to the main program.

The main program starts the same series of steps with **processB**. Upon return from **processB**, the while loop checks to make sure the master clock count limit has not been reached. If not, the loop begins again with **processA**.

4. Conclusion and Clean-up

Once the master clock count limit has been reached the program exits the loop. The final score is displayed and written to both the position data files. The processes are terminated and the program exits.

E. SUMMARY

The multiple process method was used at the expense of program portability in order to preserve the students robot codes and allow near simultaneous movement on the playing field. By using the multiple process technique the Simbots will interact with each other in a fashion nearly identical to that of the actual robots. The simulation creates the data files which are used by the graphical display or animation program. From these data files, the simulation can be designed for display in other graphic languages.

VI. ROBOT BASE

The goal of the movement simulation is to simulate proper physical reactions to the student code determined motion of the SE 3015 robots. The robot is passed a motor control string from the code to the motor controller which must decipher it and then act upon it. The robot must also handle the interrupts to its code that are caused by edge detection and the bumpers. The Simbot simulation code must perform similar tasks. Unlike the SE 3015 robots, however, the Simbots lack such problems as weight, friction and acceleration.

A. MOTOR CONTROL STRING

The robot motors are controlled by a motor control string set through the serial communications buffer. Three code lines control this operation.

1. ser_init_z1(0, 9600/1200) - sets up the appropriate serial communications buffer
2. outport(ENB485,1) - enables the 485 driver
3. ser_send_z1(buf, &cc) - sends the motor control string, buf, to the communications buffer. The string length is cc.

These code lines are found in the platform function that is general to most programs. The platform function

```
platform ()
{
        cc = sizeof(buf);
        outport(ENB485, 1);
        ser_send_z1(buf, &cc);
}
```

is provided to the student by the course instructor. The variable **buf** is an 8 character string that contains the instructions for the motors, example "ffgrffgr" which tells both motors to go forward at speed 15. Figure 6.1 provides further details on the motor control string breakdown.

Motor Control String

Robot receives a motor control string to control its direction and speed. The sting is 8 bytes long and is broken down as listed below. For example control string "ffgrffgr" indicates both motors to go, rotate, forward at speed 15.

ffgrffgr

1st byte left wheel speed (hexidecimal values 0-f corresponding to 0-15)
2nd byte left motor direction (f = forward, r = reverse)
3rd byte left motor status (g = go, s = stop)
4th byte left distance counter(generally not used)
5th byte right wheel speed (hexidecimal values 0-f corresponding to 0-15)
6th byte right motor direction (f = forward, r = reverse)
7th byte right motor status (g = go, s = stop)
8th byte right distance counter(generally not used)

The robot code sends this to an internal motor function by use of string copy to a buffer string via the **platform** function. The motor function breaks down the string as listed above and sends the direction and speed information to the left and right motors which will then turn the robot wheels. The functionality can occur any place in the robot program and as often as needed, however, if two strings are sent too close together they will "confuse" the robot and possibly cause it incorrectly execute the string.

Figure 6.1 Motor Control String Breakdown

The sending and receiving of these strings is interrupt driven on board the robot and will operate in the background while the program is doing other things. The information is passed by pointers, **buf** and **cc**, to a counter and a buffer. The serial port reads in the data one byte at a time as determined by the string length, cc, which decrements as each character of the motor

control string is passed to the motor controller. The port reads in the data until it counts down to zero. Interrupt routines change both the counter and the buffer. It takes a finite amount of time for the communications port to send the information. If a function is called before the data transfer is complete it can cause the previous platform function to be ignored and thus affect the robot's motor function, probably adversely.

B. THE PHYSICAL ROBOT

1. Robot Base Description

The robot is composed of a cylindrical body made of PVC material with two hard plastic wheels mounted at the bottom. Balance is provided by low-drag plastic bumpers at opposite sides of the robot. The robot wheels are powered by a 12 V battery and the wheel motors are controlled by the robots code. The robot code sends a control string to the motor directing the wheels to rotate forward or reverse at an incremental speed from 0-15 (Figure 6.1).

The robot base is provided to the student by the course instructor. Each student's base is nearly identical, with only minor differences due to manufacturing variability. The base is approximately 12 3/4 inches in diameter and 4 inches high. The wheels are 3 inches in diameter, set parallel to each other approximately 9 inches apart, Figure 6.2. Two rubber stoppers, each approximately 1/4 inch high are mounted fore and aft on the robot's center line to help maintain the robot balanced on its wheels.

As per the SE 3015 robot competition rules of February 1996, robot height is restricted to 4 feet tall and a weight of 35 lbs. The weight of the robot and positioning of the weapons systems on it can effect the robot performance, such as decreased speed and drag or friction

degradation the robot's performance. When the upper weight limit is reach and the robot's components are not properly centered, the robot will be resting on one of the rubber stoppers, which will impair its motion.

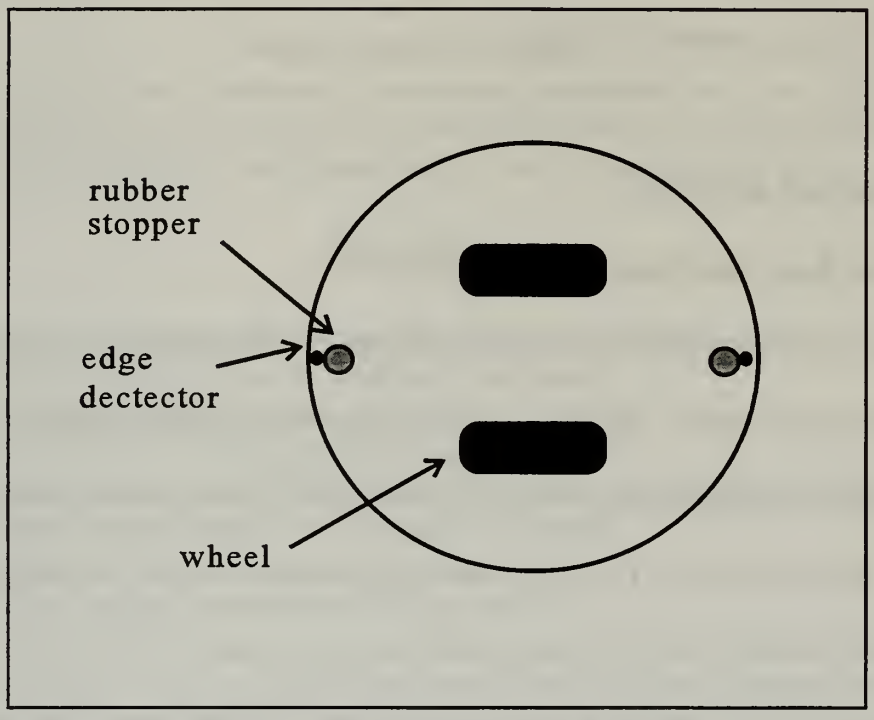


Figure 6.2 Robot base undercarriage

2. Bumpers

The robot's bumpers consist of two long contact switches mounted across the front and back on the bottom edge of the robot base (Figure 6.3). The bumper contact switches are activated when the robot runs into an object on the playing field or into the opponent robot. When activated, the robot's motor controller will interrupt, or override, any current motor-control strings and move away from the other object based on which bumper has been activated, either forward or in reverse. The bumper routine is hard coded into the robot motor controller and

cannot be changed by the student, therefore each of the robots will react in a similar fashion when a bumper is activated. If the forward bumper is activated, the robot moves backward away from the object, if the aft bumper is activated, the robot will move forward away from the object. This is discussed further in section D.4. After being bumped, the robot will move at its maximum speed, 15, for approximately 1 1/2 to 2 feet.

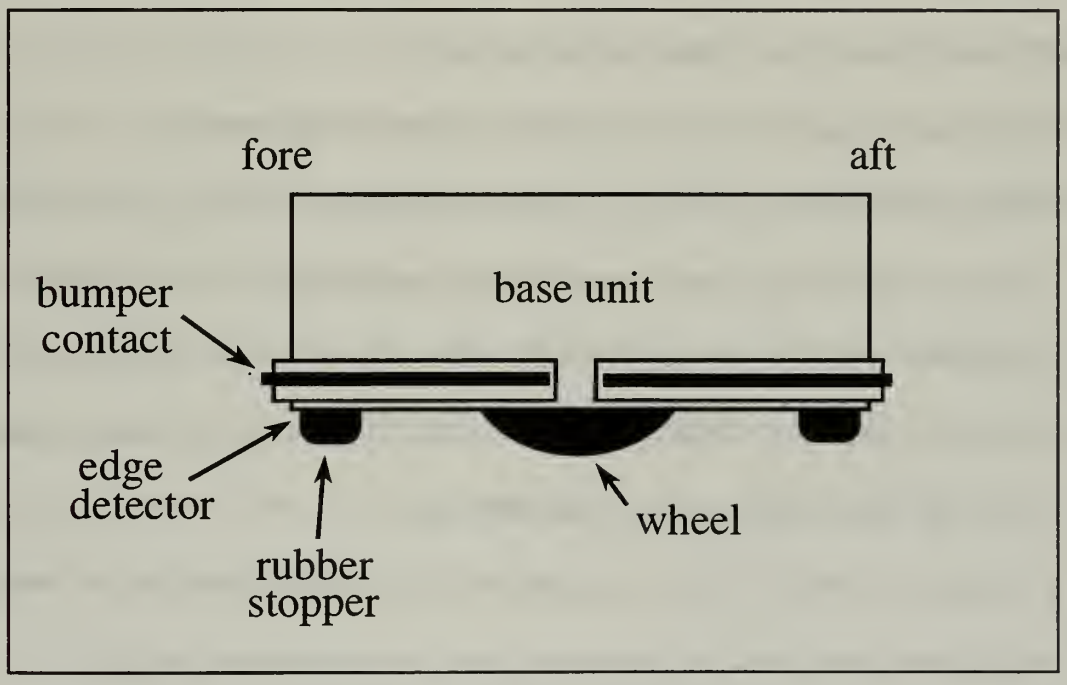


Figure 6.3 Side view of robot base

3. Edge Detection

The robot uses an electronic edge detection system provided by the course instructor. The playing field is ringed by an wire emitting a radio signal. The wire ring is set approximately 4-6 inches in from the edge of the playing field on all sides. The robot has two small edge detectors mounted on the underside of the base near the rubber stoppers, Figure 6.2, that will detect this signal. The detection circuitry, although provided and mounted inside of the

base must still be fine tuned with a sensitivity trimpot. It is possible for the student to set the sensitivity to a point where the robot will not detect the signal and run off the playing field, or to set it too high where is reacts before it physically approaches the edge.

Although each student can write their own robot's reaction to the edge detection, in general, the robot will back away from the forward edge detection and move forward away from the aft edge and perform a subsequent maneuver. The edge detection is handled by the robot's software by using interrupt flags, functions and routines.

The following pieces of code are used by the robot for edge detection.

1. #INT_VEC PIOA_VEC INT1 - is used at the start of the program to set up the interrupt handling/interaction between the hardware and software. The first part, #INT_VEC, expansion bus attention INT1 vector, directs the interrupt to a user function in Dynamic C. PIOA_VEC refers to the PIO parallel port channel A and INT1 is the expansion bus attention line interrupt.
2. interrupt reti INT1() - is for a user defined Dynamic C interrupt function that the first code line vectors the interrupt to. The robot's reaction to the edge detection on the playing field is student programmed.
3. inport(PIODA) - this I/O command returns an integer value that indicates whether the forward or aft edge detector has been triggered.
4. EI() - is a Dynamic C library function call to reset the interrupt vector.

As the robot code is running, it is continuously checking for the interrupt signal. The code lines are placed at different places within the robot code and are essential to the proper handling of the interrupt function.

4. Calibration Problems

The robot wheels theoretically rotate at the same rate when given the same speed and direction commands. However, students have observed on occasion that one wheel moves at a slightly different speed than the other, with the difference speed dependent. It is possible to use the built in trimpots in the robot base to make the wheels rotate at the same rate for a certain speed but this cannot be achieved for all speeds. If at the highest speed, 15, the robot curves slightly to the left, the student can adjust the robot's trimpot until the robot moves in a straight line at this speed. Unfortunately, the student will then observe the robot curving slightly to the right at lower speeds. The difference in speeds can be compensated for by adjusting the robot motor control strings to eliminate long movements.

The change in speeds from -15 to 15 in each wheel is not linearly incremental. Although each speed is indicated by an integer, the difference between speeds is not. The change in speed between 1 and 2 is not the same as between 2 and 3 and speed 4 is not necessarily twice as fast as speed 2.

5. Battery

The robot uses a single 12 V battery to supply power to the motor and computer, and optionally this battery can be used to power the optical and weapons circuits. The battery must be continuously charged to maintain the full charge necessary for proper operation of the robot. If the battery becomes too low, the robot may perform poorly or not function at all.

C. SIMULATION MOVEMENT

1. Simulated Robot (Simbot)

The Simbot is generated in C++ and visualized (i.e., animated) using OPENGL/XMotif. The Simbot is scaled to the robot at 100 units in diameter and 250 units tall. The Simbot is consider perfectly balanced, and friction and acceleration are ignored. The Simbot moves in a two-dimensional playing field, grid size 1600 by 1600 units in the X, Z plane where the angle of the Simbot's rotation is measured clockwise from the Z axis, Figure 6.4. For ease in creating the initial simulation each incremental move is one time unit and each incremental move is calculated in grid units.

The Simbot wheel radius and distance between wheels was set at a ratio of 0.25 to 1. With these measurements a Simbot moving at the max speed of 15 will move 15 grid units per time unit or if rotating on axis at this speed will rotate at 15° per time unit.

2. Motion

The actual movement of the Simbot is calculated by Equations 6.1 through 6.6. The incremental angle of rotation for the Simbot is calculated via Equations 6.1 through 6.3, using the angular velocities, ω_1 and ω_2, obtained from the motor control string, the radius of each wheel, r_1 and r_2, and the distance between the two wheels. The total rotational angle, θ, of the Simbot is found by adding $\Delta\theta$ to previous total angle θ. For the purposes of this initial simulation, the radii of the wheels are assumed the same.

$$v = \omega r \qquad (6.1)$$

$$\Delta\theta = \frac{2 * (v_2 - v_1)}{d} \qquad (6.2)$$

$$\theta = \Delta\theta + \theta_0 \qquad (6.3)$$

The linear velocity of the robot is calculated by Equation 6.4.

$$v = \frac{v_1 + v_2}{2} \qquad (6.4)$$

The incremental movements in the x and z plane are calculated from Equations 6.5 and 6.6 using the total rotational angle θ.

$$\Delta x = vt \cos\theta \qquad (6.5)$$

$$\Delta z = vt \sin\theta \qquad (6.5)$$

3. Bumpers

Currently, the robot wars simulation is not designed for objects/obstacles on the playing field other than the two robots. A hit on the bumper and the subsequent reaction has the highest priority on the robot's operation. When moving the robot must respond to a bump before an edge detection and before acting on the current motor control string instruction. The response to the bumper is essentially the same for each of the robots and the response is not programmable by the student, as can be done for edge detection.

The bumpers are simulated by checking one Simbot's location on the playing field relative to the other Simbot. Since each Simbot is 100 units in diameter, if the range between the two Simbots is less than 100 units, then the Simbots have "bumped" into each other. As part of calculating its position on the playing field, the Simbot checks to see if it has run into the other Simbot. At that point it will know that it has hit the other Simbot with its front bumper if it is moving forward or its aft bumper if it is moving backward. The simulation will then pass a flag to the other Simbot to indicate that it has been bumped. The relative direction of the guns of the two Simbots will determine which of the opponent Simbot's bumper has been hit, as shown in Figure 6.4 and 6.5. If the two robots are facing each other or both are facing away from each then both will have hit the same bumper, both forward, Figure 6.4, or both aft bumpers. If one is facing opposite the other then the opponent Simbot will have been hit on the opposite bumper, as shown in Figure 6.5. Once bumped, both Simbots will begin to move away from each other in a direction opposite the hit bumper.

If the bumper of a stationary robot is hit, the robot will move away from the hit at its maximum speed for about a foot. This translates to a speed of 15 for a count of 10 in the simulation . In the case of the impact of the two robots, the effects of a collision have to be accounted for. For the simulation, the collision will be considered elastic with both robot of equal masses. If the collision are perfectly elastic, both Simbots after collision at velocity 15 would each move away at velocity -30, however, the maximum velocity of the robot is limited by the motors. In the simulation, the speed of the bumper routine is limited to 15. Full analysis of the collision effects on the bumper routine is left to future research.

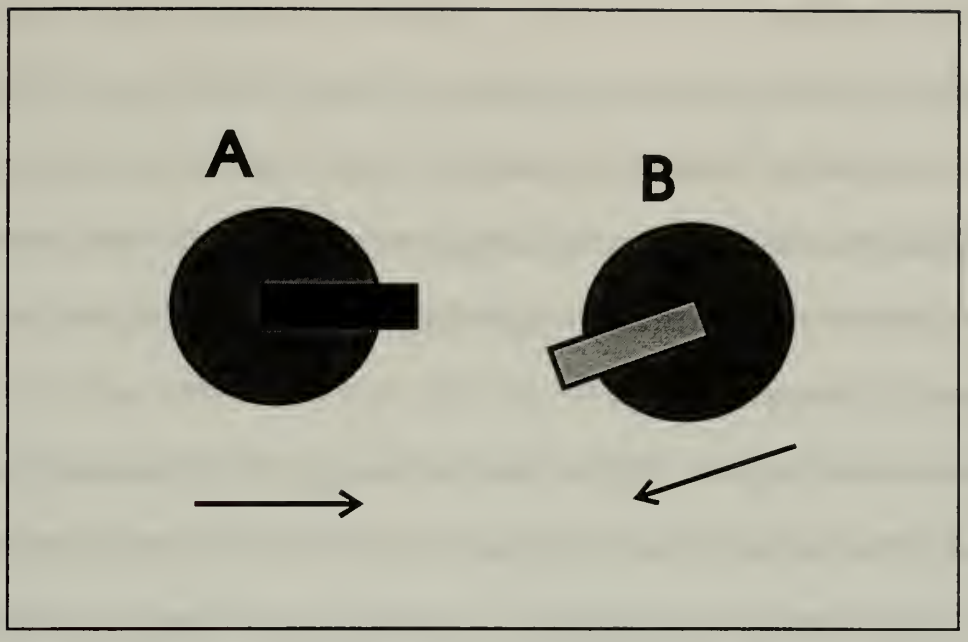


Figure 6.4 SimbotA's fore bumper will hit SimbotB's fore bumper. After impact both Simbots moves backwards. Arrows indicate direction of travel before impact.

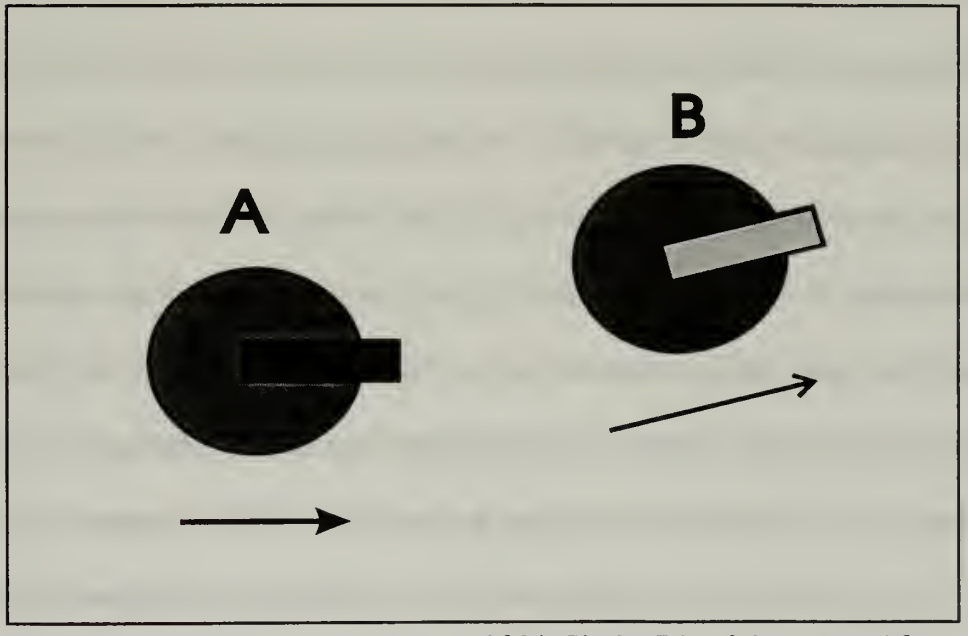


Figure 6.5 SimbotA's fore bumper will hit SimbotB's aft bumper. After impact, SimbotA moves backwards and SimbotB moves forward. Arrows indicate the direction of travel.

4. Edge Detection

The Simbot will detect the edge of the playing field in a manner similar to the actual robot. It is when the edge detector crosses edge of the playing field that the robot will detect an edge and respond according to the response programmed by the student. Robots may have different responses to both the fore and aft edge detectors. Generally a robot will respond in a fashion similar to its bumper response.

The most-used version of the edge-detection interrupt routine in the actual robot wars is for the robot code to immediately enter an edge-detection routine after the edge has been detected. In this case the robot will interrupt its current movement and move in accordance with the preprogrammed response. A typical response to the edge detection is: for aft detection to move forward away from edge and for fore detection to back up and turn to face away from the edge.

The simulation 1600 x 1600 unit playing field has an edge detection ring 50 units in on all sides, reducing the playing field to 1550 x 1550 units, see Figure 6.5. The forward and aft edge detectors are points on the central axis at the front and rear of the robot 5 units back from the edge of the Simbot. The edge detection sensitivity can be simulated approximately by changing the location of the edge detection points. Moving the edge detection points closer to the center of the Simbot, the Simbot not detecting the edge until it is almost on it. Moving the edge detection points outside the Simbot edge, will make the Simbot more sensitive to the edge. As with the robot, speed and other factors such as interrupts from the bumpers can cause the Simbot to fail to detect an edge.

D. AUTOMATED REFEREE (AUTO REF)

Although most of the robot's problems dealing with motion are ignored or avoided, the simulation experiences complications that affect the actual robots. The Simbots are capable of going off the playing field by ignoring an edge-detection or getting hung up, either alone or in pairs, on the playing field edge. As the simulation is run in C++ without visualization, an automated referee has been added in an attempt to get the Simbots back on the playing field.

The Simbot is generally more reliable than the real robot in detecting the edge but such things as a bumper interrupt may cause it to run off the playing field. If this occurs the auto ref will place it back onto the field near where it left. This correction is determined in the function **checkposition( )** which act like an edge detection and checks to see that the Simbot has not run off the field and makes the corrections as previously discussed. When placed back on the playing field, the Simbot can reevaluate the edge detection and then respond accordingly.

The case of the Simbot getting "hung up" on an edge is more complicated as it may involve the other Simbot. Both the edge detection and bumper interrupt functions use a separate counter to determine the length of the Simbot's response. If another edge or bumper is encountered before the first routine is complete, the Simbot will start the new routine and will set an internal flag to indicate that multiple interrupts are occurring. The **auto_ref( )** function checks for more than three bumper routines or a combination of more than five bumper and edge detections in succession. If this criteria has been met, **auto_ref( )** function will alter the Simbot's directions slightly to break them out of the multiple interrupt routines.

Interrupt hang-up situations that will not be corrected by this function are where the Simbot exits the bumper or edge detection routine completely and then immediately enters the

routine again. In order to achieve this situation with the bumpers, both Simbots, since there are currently no obstacles on the playing field, must approach each other on a straight path at angles 180° apart and be"locked" into that same path (moving back and forth toward each other). However, since ramming the other robot is not a scoring possibility and is not a recommended tactic, this situation although possible is deemed unlikely and ignored. The second case is the Simbot "walking" along the edge. During the SE 3015 Robot Wars Competitions held over the last two years it has been observed that the referee has not interfered in this case unless the robot rolls off the edge or becomes stuck in one location. Based on that observation, the case of "edge walking" will also be ignored as this scenario is generally "corrected" when the robot enters a corner, observes the other player, or is bumped.

E. SUMMARY

The simulated robots will compete on a playing field scaled to the field of the SE 3015 robot wars competition. With friction and acceleration ignored, the simulated robots still lack the realism found with the SE 3015 robots. The simulated robot wars is designed at this point to give the student a feel for how their robots will respond to its programming. Aided by an automated referee, auto ref, the simulated robot competition will be able to handle some of the basic movement problems encountered by the SE 3015 robots.

VII. OPTICS

The robot uses a pair of photo detectors (refered to as "eyes") to see the beacon on the opponent robot. The eyes are connected to a pair of circuits tuned to detect the beacon frequency of the opponent robot and return a value corresponding to the amount of light detected. The two eyes are mounted above the robot's own beacon and are set with a divider (referred to as a "nose") between them which permits left and right angular discrimination. The angle at which the robot sees its opponent will determine the amount of light reaching the eyes and values returned.

A. BEACON

In order for the robots, to detect each other each are equipped with an LED beacon that transmits a 360° signal at a unique frequency. The robot optical circuit is designed to detect the assigned frequency of the beacon. The current SE 3015 beacon, Figure 7.1, emits light in a uniform 360° distribution pattern in the horizontal plane. The beacon transmits a set frequency with an approximate 10° vertical spread. Each robot has its own beacon frequency and the optical circuit is tuned to detect this frequency. The robot's beacon is placed on the opponent robot for detection at the time of the competition.

The beacon is mounted underneath the center of the top deck disk, 4.56" above the top of the robot base. The photo-detectors are mounted on the top of the top deck above the beacon. The beacon is mounted so that only the four thin cylindrical platform support posts obstruct it. These four support posts provide minimal obstruction to the beacon signal transmission and are considered a negligible obstruction and are ignored in the simulation design.

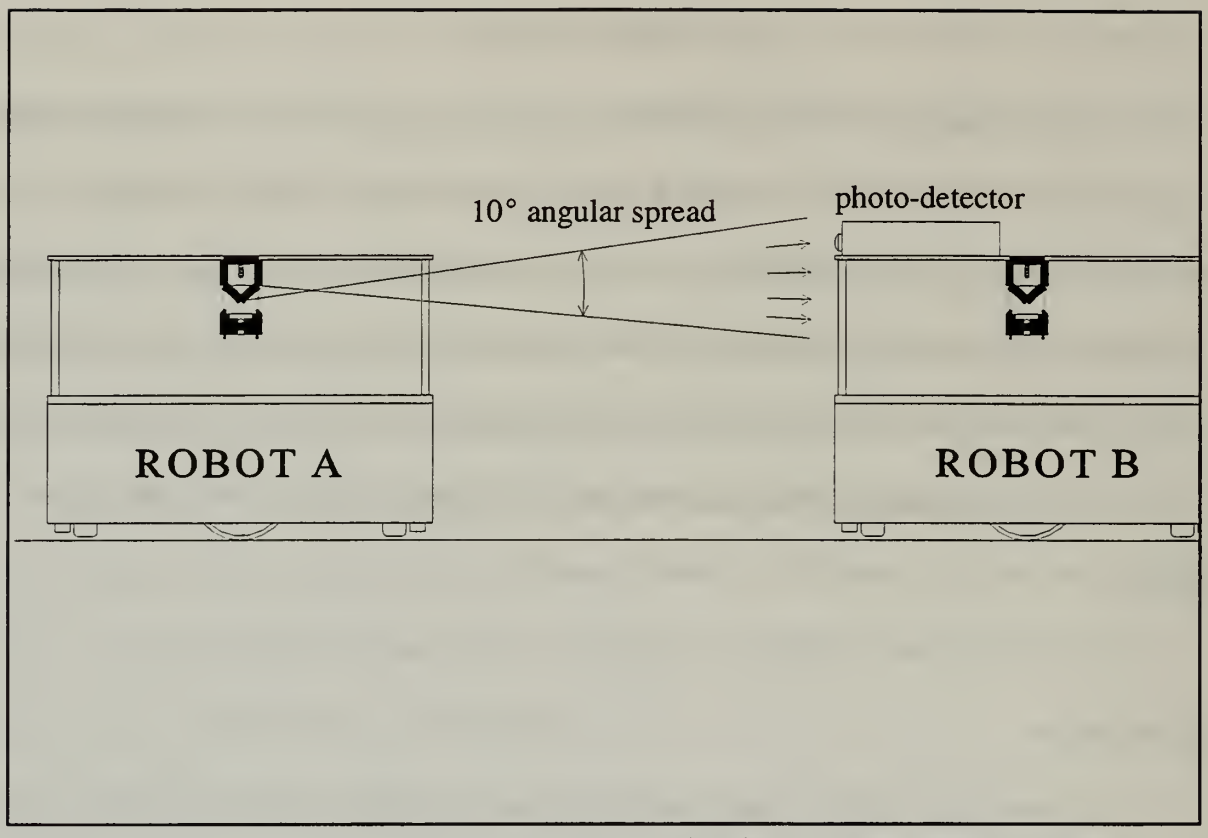


Figure 7.1 Side view of robot with beacon in place showing 10 degree angular spread (Hofler, 1997)

B. THE PHYSICAL EYES

The robot optical detection systems are designed to detect a specific frequency for which they have been tuned. These eyes will detect the frequency transmitted by the beacon on the other robot and determine the intensity of the received signal. The circuit is designed for a high maximum for the Q value to minimize the noise and interference caused by other light sources.

The eyes are made up of a pair of photo-detectors, mounting apparatus and a pair of optical detection circuits. The photo-detectors used have an active area of 0.17 cm^2. They are mounted with a divider plate between them to allow for discrimination between the left and right eyes (Figure 7.2). With this divider, the eye on the near side will receive the full signal, but

the other eye will only receive a portion based on its viewing angle. See discussion on viewing geometry appears in Section C.

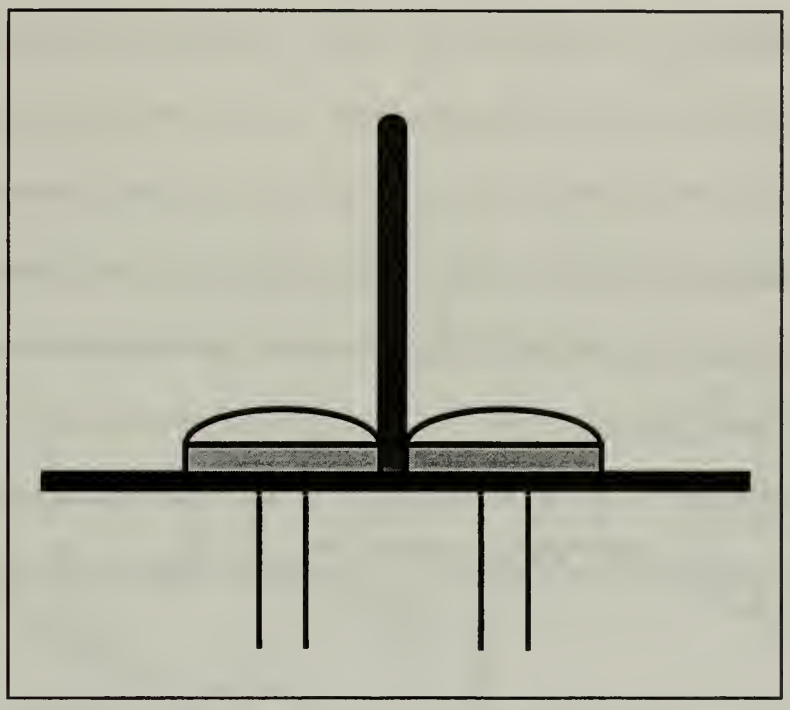


Figure 7.2 Photo-detector pair with nose

Interference and noise contribute to detection errors. An AC-coupled RC-bandpass filter circuit, Figure 7.3, minimizes the interference caused by the fluorescent room lights. Additionally, each eye may have its own separate circuit to minimize cross-talk interference between them. The circuit design allows for a relatively high Q , resulting in the robot easily detecting the opponent within the confines of the 16 x 16 foot playing field (as shown in Figure 3.1). Due to the high sensitivity, the eyes become saturated when the beacon is closer than a few feet.

The eye circuit is generally powered by two nine-volt DC batteries, which must be fully charged for proper performance. As the batteries begin to drain, the Q can decrease giving erroneous detection values due to reduced discriminating of the beacon signal from other external light sources.

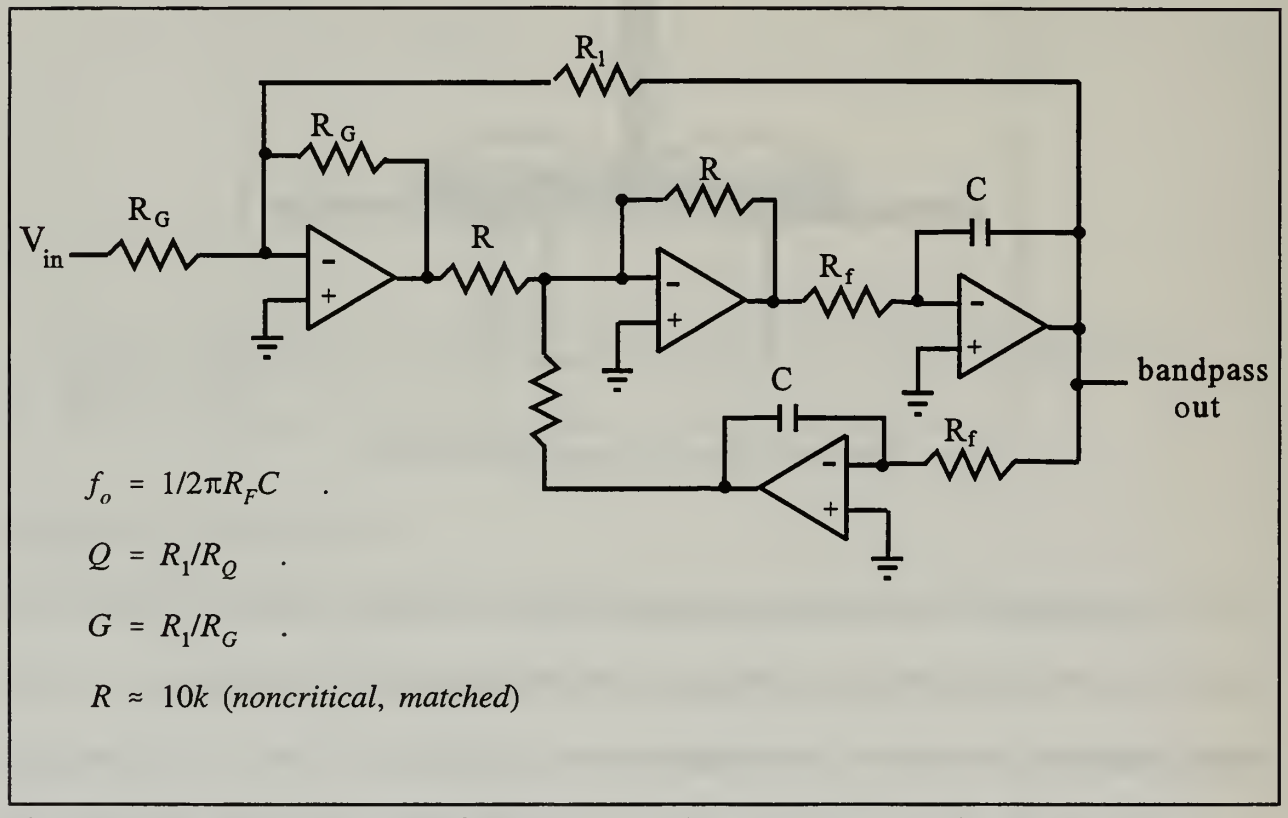


Figure 7.3 Optical circuit for SE 3015 robots. This is "a useful variant of the state-variable bandpass filter. The bad news is that it used four op-amps; the good news is that you can adjust the bandwidth (i.e., Q) without effecting the midband gain. In fact both Q and gain are set with a single resister each. Q, gain and center frequency are completely independent and are given by these simple equations" listed above (Horowitz, 1989, p.278)

C. VIEWING GEOMETRY, LEFT AND RIGHT DISCRIMINATION

The capability for left and right discrimination is one of the more crucial features of the robot's optical system since it is required for targeting. The robot cannot know where it opponent is on the playing field relative to itself without being able to make this discrimination. The left and right discrimination allows the robot program to determine both the approximate distance of its opponent on the playing field and where its opponent is in relation to itself. In order to produce discrimination, a divider is placed between the photo-detectors to limit the directions of light reaching each eye. Without this, both eyes would receive the same amount of light regardless of where the robot was within the 180° viewing area. The ratio of the nose height to the width of the eye determine the limiting angle, or cut off values, for left and right discrimination. The eye widths are measured from the outer edge of the photo- detector to the nose for each side (Figure 7.4).

The limiting angles, Equations 7.1 and 7.2, are calculated using the arc tangent function, where α is the limiting angle for the left eye and β is the limiting angle for the right eye.

$$\alpha = \arctan \frac{w_r}{h} \qquad (7.1)$$

$$\beta = \arctan \frac{w_l}{h} \qquad (7.2)$$

The longer the nose is the smaller the limiting angles. With large limiting angles and the robot will not be able to discern left from right, and if the angle is too small the robot will only be able to see out of both eyes when the opponent is directly in front of it.

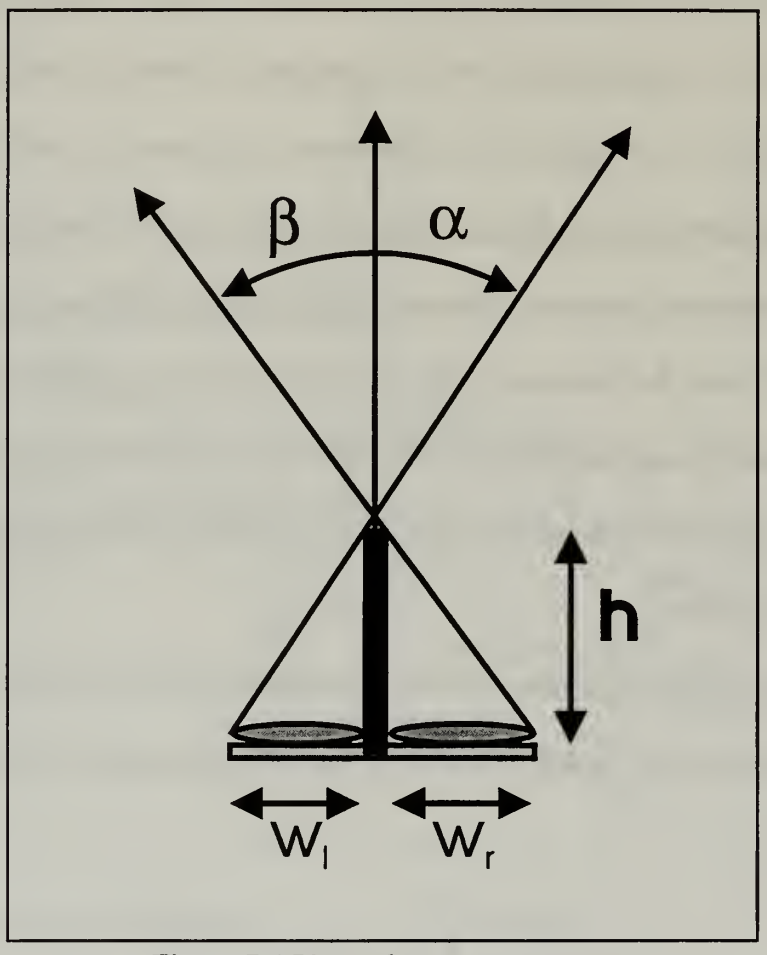


Figure 7.4 Photo-detector geometry

The opponent (Figure 7.5) is located on the left side of the robot and thereby in full view of the left eye. If angle from center line, γ, is less than the right limiting angle, β, then the right eye will be able to detect the opponent as well. The fraction of the photo-detector active area is determined by Equation 7.3, where γ relative angle of the Simbot's gun to the opponent and ϕ is the limiting angle (α or β depending on which side the opponent is on). This fraction is

corresponds to the amount of light from the beacon that falls on the partially obscured eye. The full signal level is multiplied by this fraction to obtain the simulated signal level of the partially obscured eye.

$$fraction = 1 - \frac{\gamma - \phi}{\phi} \qquad (7.3)$$

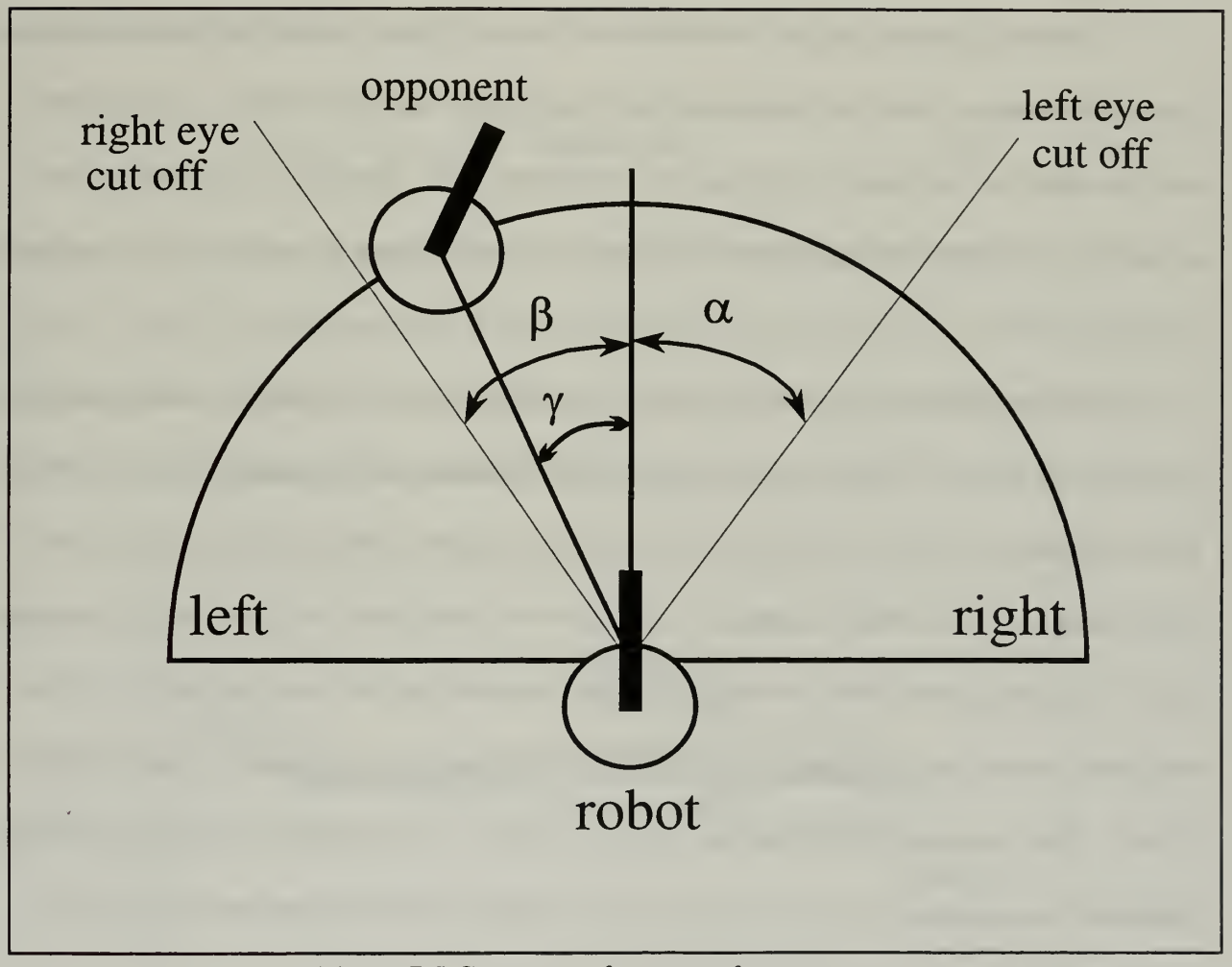


Figure 7.5 Geometry of sensor reference system

D. SIMULATION OPTICS

Although the optical circuit uses the specific gain, Q and detection limits for the robot, the simulation only uses the detection distance and the maximum detection values are used in . The simulation has essentially an infinite Q (i.e., there is no sensitivity to range). The placement of left and right eyes are not considered identical as they have a direct impact on the robot's targeting ability.

In order to generate this first simulation of the robot's eyes, several basic assumptions on their performance will be made concerning the optical system and the beacons. The simulation does not allow for interference, electrical noise or battery drain. However, the simulation user will still have to provide the saturation value, saturation distance and dimensions of the nose and eyes.

After completing the eye circuit and while designing the robot code, the student considers the robot maximum eye value obtained and the physical distance for saturation of the eye circuit. The maximum eye value is a function of the circuit and is either 2040 or 4080 depending on the circuit design. The saturation distance for each robot is a function of the Q and is approximately 3 feet. The background value observed by the eyes when the beacon is not in sight is 16. The maximum distance that the robot needs to see is the diagonal of the playing field, approximately 22 feet. These are the initial values used in this simulation. The Simbot class function **setEyes**, presented in Appendix C, allows for the eye values and photo-detector geometry to be set for each of the Simbots.

In order for the Simbot eyes to function similarly to the robots eyes, the photo-detector geometry has to be determined and entered into the simulation using the **setEyes** function. The

three measurements should be to the same degree of accuracy and must be in the same units. The simulation use the measurements in a ratio and the specific units are not important.

The LED used in the beacon, described in Section B, distributes light spherically, Figure 7.6. The beacon housing used, limits the light to a 10° vertical spread. A theoretical spherical distribution with a $1/r^2$ fall off is used to simulated values. The amount of light seen by the detector is dependent on its field of view and the solid angle of the light from the beacon as represented in Equation 7.4.

$$P_{detector} = P_{source} \frac{\omega}{\Omega} + background \qquad (7.4)$$

Where ω is the solid angle on the detector and Ω is the solid angle of the light leaving the beacon. The 10° angular spread of the beacon translated to a solid angle of 1.096 steradians, very close to 1 steradian. The active area of the detector is a constant size and much smaller than the $r^2\Omega$. The curvature of the lens of the detector is ignored.

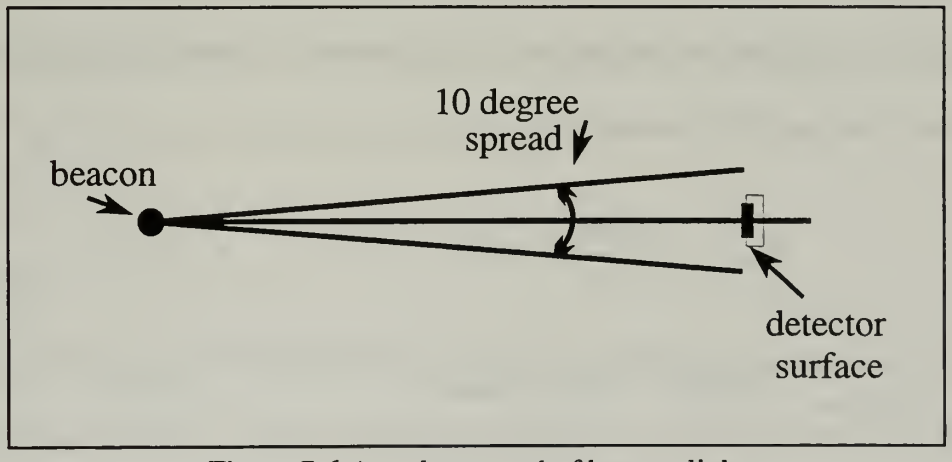


Figure 7.6 Angular spread of beacon light

Attenuation and the optical efficiencies of the LED and the photo-detectors contribute to the degradation of the eyes but will not be considered in the simulation at this time. The background value is the point where the circuit is unable to distinguish between the circuit noise, interference from other light sources, and the beacon. This value is a constant for a given circuit but may change due to the circuit design and individual circuit construction.

The above considerations lead to the following development for the eye signal levels. The spherical distribution, Equation 7.5, was selected to simulated the eye signal levels, S_{eye} that would be obtained by the Simbot's optics.

$$S_{eye} = \frac{\kappa}{r^2} + S_{background} \qquad (7.5)$$

$$where\ \kappa = \frac{P_{source}\ \omega}{\Omega}$$

The parameters of a eye signal level of 4080 at 3 ft and a background value of 16 were entered into Equation 7.5 to obtain a value of 38578 for κ. Equation 7.6 is used to determine the value for the eye in full view of the opponent and Equation 7.6 is used to obtain the theoretical value for the for the partially obscured eye.

$$S_{full\ eye} = \frac{36578}{r^2} + 16 \qquad (7.6)$$

$$S_{eye\ frac} = (\frac{36578}{r^2} + 16) * fraction \qquad (7.7)$$

The fraction value is calculated from Equation 7.1.

E. INTEGRATING EYE SIGNAL LEVELS

The SE 3015 robot receives the eye values through a function call to an I/O port through an analog-to-digital (A/D) convertor . The Simbot will receive its eye values for a similar function call. The returned value in both cases will be an integer between the minimum, (currently 16) and the maximum (4080) allowable eye values.

The robot obtains the eye values from the optical systems by a manufacturer-provided Dynamic C code function call **ad_rd8(int n)** which returns an integer value. The function parameter "int n" indicates which eye to obtain a value from. The number of eyes that the robot can have is hardware dependent. However, most students use only one set of eyes and with n equal to 0 or 1 to indicate the left or right eye respectively.

The simulation function call **botA.ad_rd8(int n, simbotClass &botB)**, also returns an integer value to indicate the intensity level seen by the simulated eye. The simulation function call however, requires additional information to obtain this value. The function call must be the specific class function assigned to the Simbot, botA, and it must be able to have access to the location of the opponent, botB. The simulation assumes only one set of eyes are in use and that the left eye is indicated by n = 0 and the right by n = 1.

F. SUMMARY

The simulated eyes provide values to the simulated robot as the optical system provides the values to the actual robot. The geometrical configuration, essential for left and right discrimination has been incorporated into the model, a critical factor for targeting of the opponent. The simulated eye function artificial ranges set at 3 feet for saturation, eye value 4080

and the eye background value of 16 and values are simulated using a spherical distribution pattern. Noise, and interference are not incorporated into this model.

VIII. WEAPONS

This section discusses the "standard gun" used by the student, alternative weapons, the ammunition used, and how they are incorporated into the simulation. The current SE 3015 course has incorporated a standard gun to "level the playing field" allowing students to concentrate on other portions of the robot design. The requirement for the use of a standard gun varies with each SE 3015 class as well as the number of weapons allowed. This section briefly discusses the possible types of weapons systems, but the simulation currently only allows for the use of projectile weapon systems.

In order to properly simulate a weapon system, its dynamics and kinematics must be fully understood. Additional considerations include simulating problems such as jammed weapon's barrel, failure to fire, and double rounds where two rounds are fired at one time without enough energy to propel them both.

Three basic types of weapons have been used the last few iterations of the SE 3015 Robot Wars Competition: projectiles, catapults and obstacles. The standard gun is a projectile weapon and will be discussed first.

A. STANDARD GUN

The standard gun, Figure 8.1, shoots a 0.5" wooden ball from a solenoid triggered gun. This weapon was first used in March 1996, to help equalize the weapons component of the competition and allow the student to concentrate on other robot components. The next iteration of the SE 3015 course allowed students to use other weapons in addition to the standard

gun. Multiple weapons types have been allowed, but the simulation currently allows only for simple projectile weapons (ie., the standard gun).

The standard gun incorporates a barrel, wheels, motor, solenoid trigger, firing pin and magazine barrel. The gun is initially activated by turning on the motor that drives the wheels. These wheels spin at a high rate and will propel the projectile when pushed between them. The gun is gravity fed and uses a solenoid to push the firing pin forward, propelling the projectile forward through the spinning wheels. The wheel motion propels the projectile forward with sufficient velocity to easily fly the full length of the playing field (22 feet diaganal).

Specific modelling of the standard gun used in SE 3015 has been deferred to later research. As with many aspects of the SE 3015, the students can tailor the performance of the standard gun to their requirements. The wheels that propel the projectile can be power by 6V DC to 10 V DC. The voltage used will determine the specific speed of the projectile.

The standard guns can malfunction in several areas. The wheels which propel the projectile out of the weapon use O-rings. This O-rings can lift off of the wheels if the wheel speed exceeds its tolerance. This occurs when voltage above 10V DC are used. The gun is gravity feed from the magazine tube. The gun in the ready position will have round sitting in front of the trigger, and the solenoid firing pin assembly and behind the wheels, Figure 8.1. If the firing pin does not push the projectile into the wheels with enough force it will not catch on the wheels and remain in the chamber. This misfire will allow a second round to drop into the chamber and cause the projectiles to jam or to fall out since there will not be enough force to push two out of the chamber on the next trigger fire.

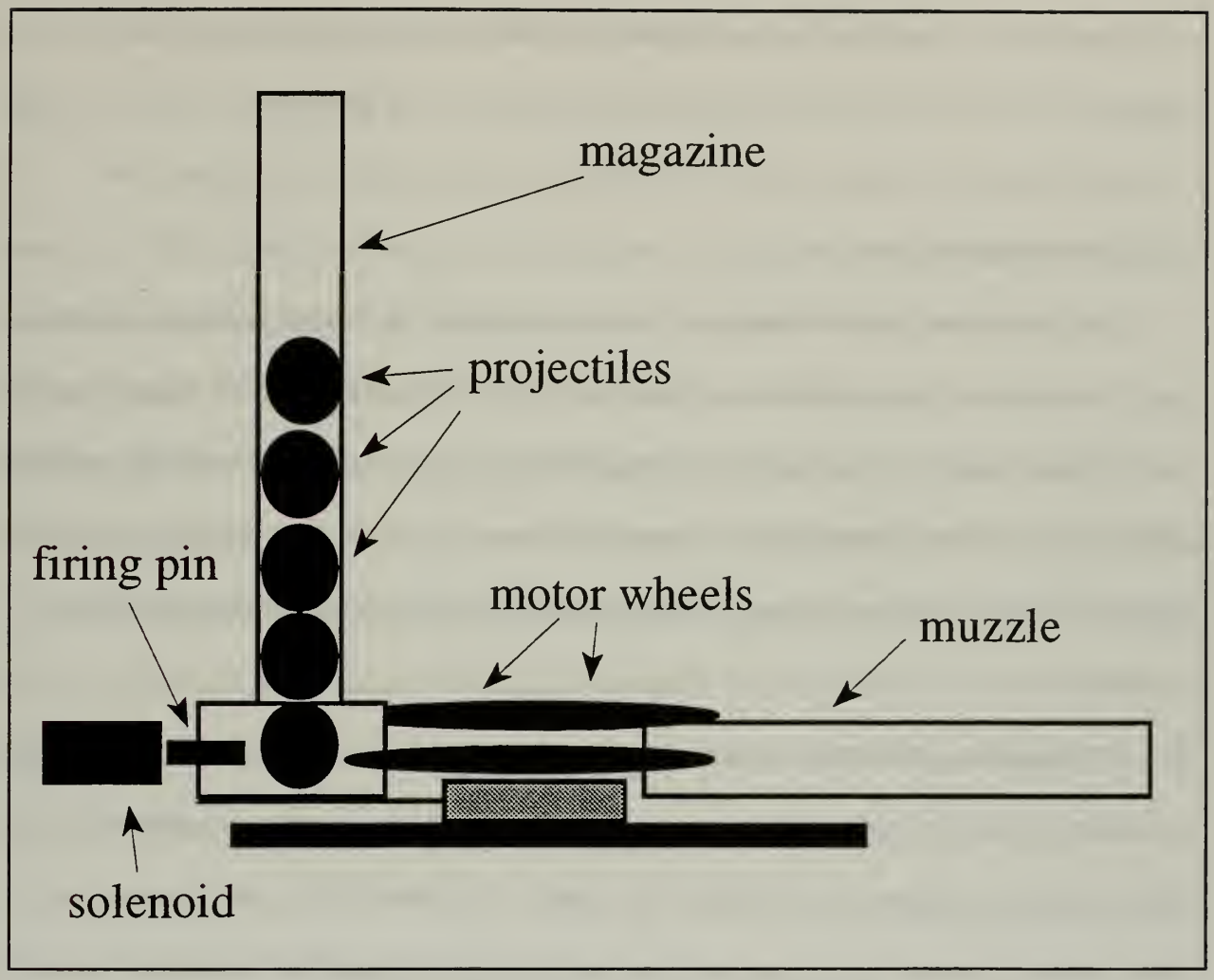


Figure 8.1 Standard gun side view block diagram

B. ALTERNATIVE WEAPONS

Additional weapons systems used on the SE 3015 are only limited by the imagination of students, the weight and power supply, muzzle velocity and a few safety restrictions. A weapon must be safe enough for a person to take a hit. High speed projectiles, explosives, and NBC weapons (except Silly String) are not permitted. Weapons systems in the past have shot flying

discs, and pencils in addition to nets. Nets and Silly String hits are not scorable in the competition and are rather used to sabotage the functionality of the oppoenent.

C. SIMBOT WEAPONS

Despite the use of the standard gun there is not a full set of standard parameters for these guns. The standard gun's muzzle velocity will vary depending on the amount of voltage used to drive the wheel motors. It is possible, although difficult, for the student to calculate this velocity and enter it into the simulation, but this has not been done at this time. The velocity used in the simulation is calculated from the projectile motion and observed behavior of the standard gun projectile.

The simulation uses only an estimated exit velocity for the standard gun. The gun muzzle is considered to be flush with the edge of the robot. The possibility of weapons malfunction from ammunition jamming or ring slippage is ignored. The solenoid firing pin is assumed to always hit the projectile with enough force each time to push the projectile through the wheel to launch the projectile.

D. PROJECTILES

The basic types of projectiles categories used in the SE 3015 course have included small spherical projectiles, odd-shaped projectiles, and soft objects. The spherical projectiles have included the wooden ball used in the standard gun, gum balls used in an air compression gun, and ping pong balls. The odd shape category includes a wide range of projectiles from pencils to

foam discs varying in density, size, and shape. The last category (soft objects) includes such things as nets, and silly string.

For simplicity, only the standard projectiles will be incorporated into the simulation at this time. The simulation of other objects, although not impossible, requires a greater knowledge of their dynamics and interaction with the robot, as well as the behavior of their weapon system. The projectile used for the simulation is modelled from those used in the standard gun, a wooden ball with a maximum diameter of 0.516" (or 1.3 cm), with a density of 450 kg/m^3 (Kinsler, 1982, p.461) and a mass approximately 0.5 grams, assuming the projectile to be made of pine. Based on these figures, the projectile fired from approximately 1.5 feet off the ground will not reach terminal velocity, subsequently air drag is ignored. For the current simulation, the projectile motion has been simplified and ignores such things as the effects of air drag and the true initial velocity of the projectile.

E. CALCULATING DEFAULT EXIT VELOCITY

Due lack of standardization in the weapons system, it was decided to create a default velocity for the projectiles used. This default velocity was fixed by working the projectile problem backward using some assumptions of the robot construction, height of the gun, and the observations that the projectile travels the length of playing field before impacting the ground.

The gun fires horizontally from a height between 1 1/2 and 2 feet. Using the basic equations for projectile motion and considering the projectile to be a particle the total flight time for projectile is dependent only on the launch height. Taking the height to be 1.5 feet, the flight time is calculated from Equation 8.1.

$$t = \sqrt{\frac{height}{g}} = 0.21s \qquad (8.1)$$

The horizontal velocity of the projectile does not preclude the effects of air drag but it can be ignored for this approximate calculation for the standard projectile. The initial velocity of the projection has only a horizontal component. Using the assumption that the projectile will travel the length of the playing field 22 ft (6.7 m) before it reaches the ground the velocity is computed from the Equation 8.2, using t from Equation 8.1.

$$v = \frac{range}{t} \qquad (8.2)$$

This yields a minimum initial velocity for the projectile of 105 ft/s if the projectile is to reach the end of the playing field before hitting the ground. In reality the wooden ball projectile can easily exceed the 22 feet distance and therefore be travelling faster than 105 ft/s . However, until more measurements are conducted on the weapon systems used for the purposes of the simulation, this assumption should hold for most light-weight spherical objects used assumed velocity as projectiles in the SE 3015 Robot Wars competition.

F. PROJECTILE MOTION

The current simulation uses basic three dimensional projectile motion ignoring air drag, dispersion, and projectile shape. The projectile motion is broken into x and z components in the horizontal and the vertical y component. Although the y position is calculated, the projectile

used in the simulation have been set with a velocity which will preclude it from impacting the ground before leaving the playing field.

The path of the projectile is calculated from Equations 8.3 through 8.5. where θ_λ is the gun angle of the Simbot at the time of fire and x and z are the coordinates of the Simbot center of mass. The height of fire is 1.5 ft or 150 grid units. Gravity is scaled to simulation space as 320 grid units/ time unit^2

$$x_p = v\ t\ \cos\theta_\lambda + x_{p_o} \quad . \qquad (8.3)$$

$$y_p = -gt^2 + y_{p_o} \quad . \qquad (8.4)$$

$$z_p = v\ t\ \sin\theta_\lambda + z_{p_o} \quad . \qquad (8.5)$$

For the standard gun projectile, the projectile height at impact will not be a consideration. With lower velocity projectiles, it is possible for the projectile to impact the robot at the same time as it impacts the ground. At 105 ft/s, or 1050 grid units/time step in simulation space, the projectile will be able to cross most of the playing field before the opponent moves hence the subsequent movement of the opponent will be ignored.

G. SIMULATION PROJECTILE FLIGHT PATH

The simulation projectile's flight path is calculated for each time step and compared to the position of the opponent. If the projectiles flight path intersects with the body of the opponent before it reaches the ground it will constitute a hit. Otherwise it is considered a miss.

Two matrices are used to simulated the projectile motion, the gun and bullet matrices described in Section 3.C and Tables 3.6 and 3.7. The gun matrix controls which weapon will be

fired. The current simulation can handle multiple simple projectile weapons through the gun matrix but is only set for one active weapon.

The weapon is fired when the simCode calls the Simbot class **outport** function **bot.outport(register, data_value)** in its main code. This function call can be used to access different types of I/O devices by indicating the register. Most of the weapons functions are handled through the outport function and POIDB I/O port. The data value sent indicates which weapon to fire. The Simbot class outport function is currently set up to handle only the PIODB register and one weapon.

The robot class gunfire function will check the gun matrix for ammunition availability and if available will activate a bullet from the bullet matrix, assign an exit velocity, gun number and launch height to the bullet matrix elements from the gun matrix. The active bullet then receives the current x and z position of the robot, and its gun angle.

The simulation fires the projectiles from the gun, activating the simulated bullets and tracks them through their active flight with the robot class projectile function. Bullets are considered active once they have been fired until they are deactivated by impacting the opponent, the floor or by leaving the playing field. Unfired bullets are by definition inactive. The bullet's trajectory is tracked through the position function. Before the Simbot moves it checks the bullet matrix for active bullets. If an active bullet is found, the Simbot class function **projectiles( )** will calculated its next position.

In calculating the bullet's trajectory, the **projectile( ) function** first determines the range between the current position of the projectile and the opponent and compares this to the length of the bullets projectile path travelled during the time step as determined by the velocity, or *vt*. If

the range, r, of the bullets current position to the opponent is less what the bullet would travel unimpeded during the time step, the function computes the bullets new position, using Equations 8.6 through 8.8.

$$x_p = r \cos\theta_\lambda + x_{p_o} \quad . \qquad (8.6)$$

$$y_p = -gt^2 + y_{p_o} \quad . \qquad (8.7)$$

$$z_p = r \sin\theta_\lambda + z_{p_o} \quad . \qquad (8.8)$$

The Simbot then checks the height of the bullet at this new position. If the height has fallen below 0, the bullet would have impacted the floor before it reached the opponent. At this point the bullet is deactivated and scored as a miss. If the bullet is still above 0, then the Simbot calculates the distance between the bullets new position and the opponent body center. If this distance is less than the radius of the opponent, then bullet is scored a hit and is deactivated. If the distance is greater than the radius, the bullet is scored a miss and is deactivated, since it will travel past the opponent on the next time step. Figure 8.2 provides a diagram of a bullet fired from the Simbot that misses its opponent. If the range to the opponent is greater than the distance the projectile would travel during the time step, the **projectile** function will use Equations 8.3 through 8.5 to update its position. As part of updating its position, the Simbot will check to make sure the bullet has not left the playing field. If the bullet has left the playing field, it will be scored as a miss and deactivated. The only bullets that will remain active after these calculations and comparisons are made are the ones that remain over the playing field, above 0 and have not yet reached the opponent.

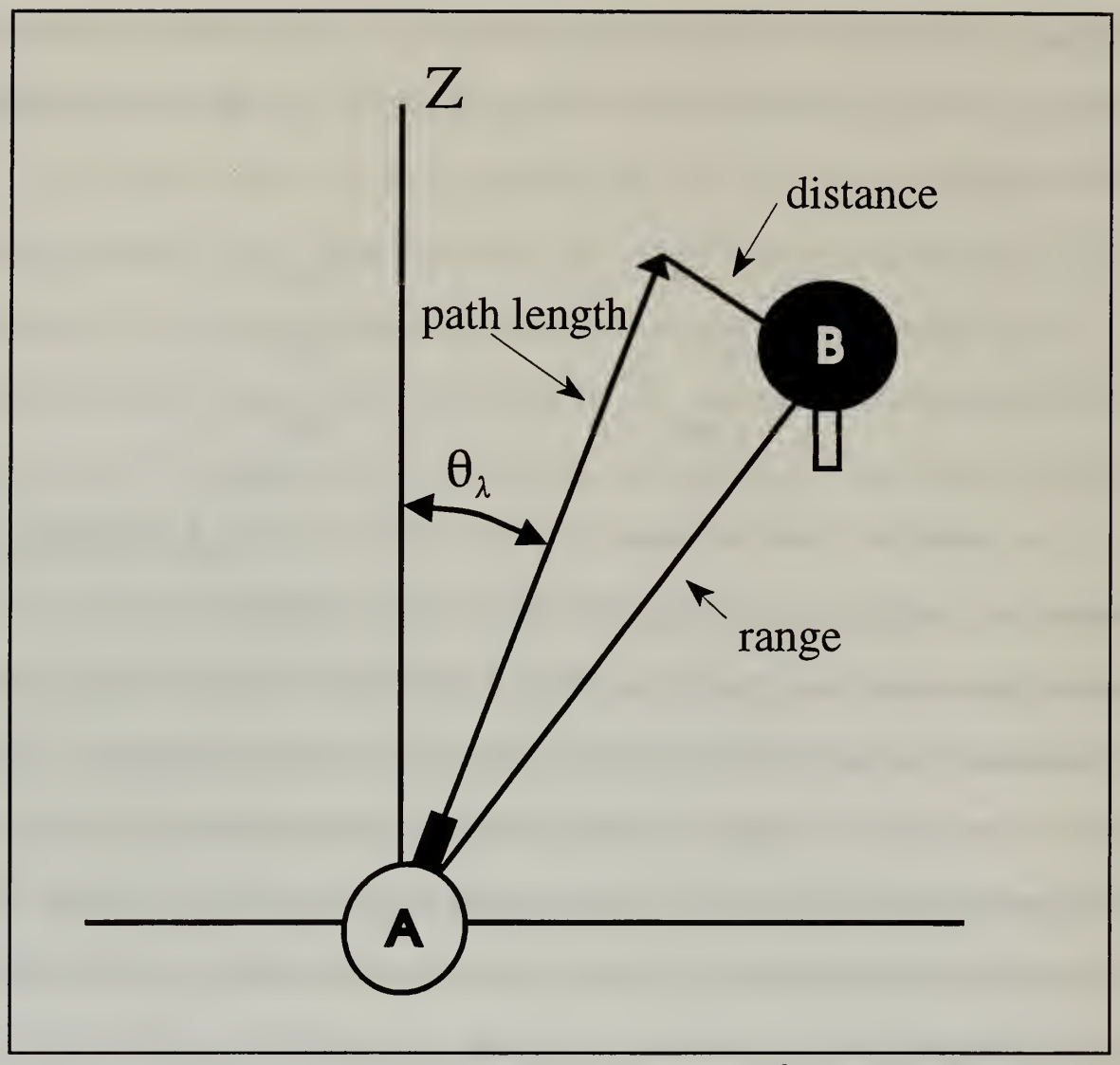


Figure 8.2 Diagram of a fired bullet that misses its opponent. θ_λ is the firing angle.

H. SCORING

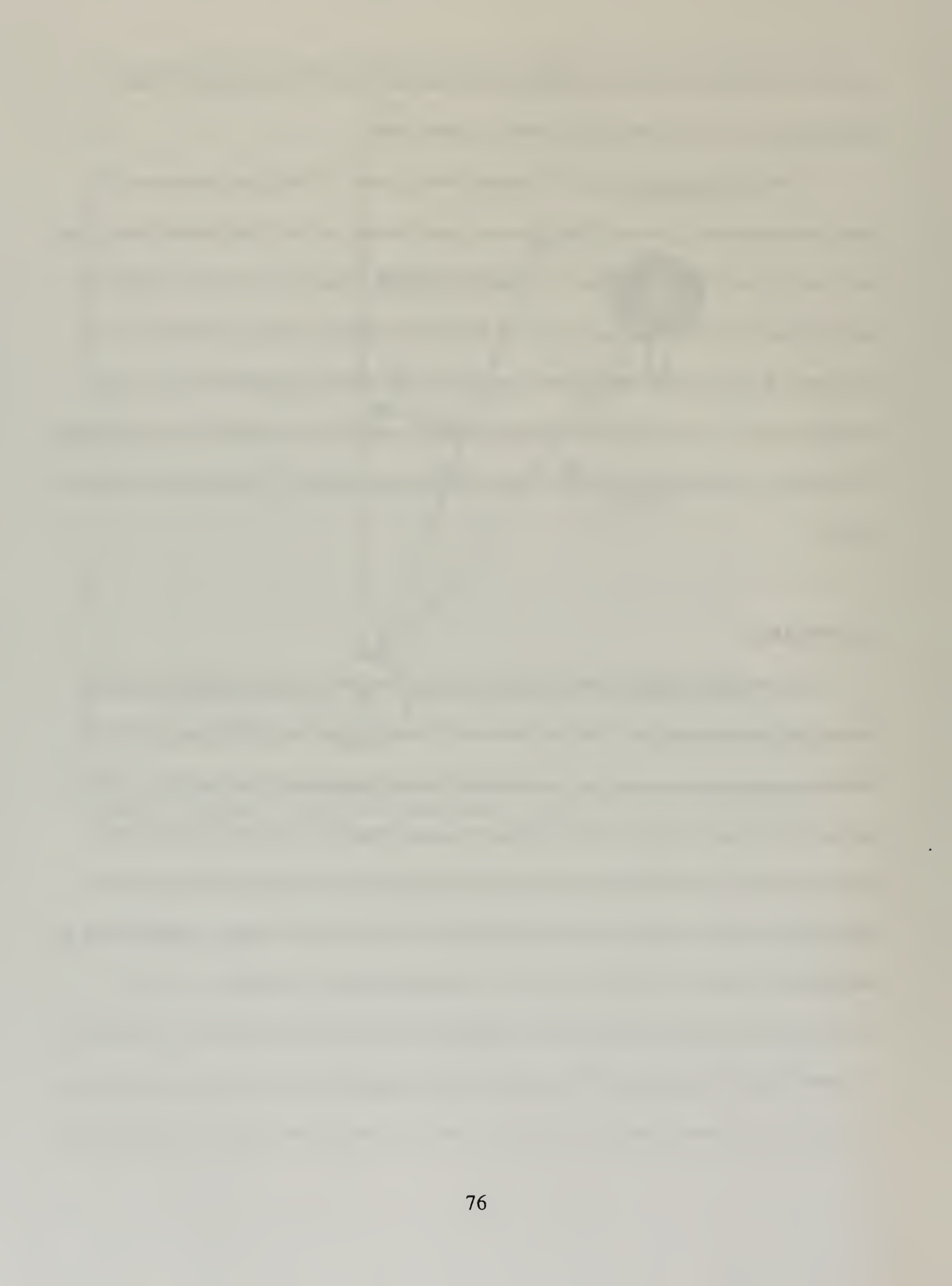
The robot which hits its opponent the most will win the Robot Wars Competition. The competition is scored by professors and/or students in the curriculum using visual and auditory cues to determine a hit. Due to the high velocity of some of the projectiles, the actual hit may not be observed. The scorer may use the sound of a hit or an observed ricochet off the robot to

verify a hit. Although this scoring method is fairly accurate, in the case of rapid-fire hits or background noise, it is possible for the scorer to miss a point.

A projectile is scored as a hit if it impacts the opponent before it hits the ground. The Simbot is considered to be a solid object and an impact on any portion of the Simbot is scored as a hit. Once a projectile hits the floor, it is no longer in play. Therefore a ricochet off the floor will not count as a score. The momentum of the ricocheting projectile is not considered in the simulation. If the projectile impacts the ground at the same time it impacts the robot it will be considered a miss. In the case of the acutal robot, it is possible for a projectile to passes through an open part of the opponents structure without hitting the actual body, which would be scored as a miss.

I. SUMMARY

The simulation weapons systems based on simple projectile motion modelled from the standard weapon ammunition. The simulation will ignore the technical difficulties associated with the real gun such as a weapons jam will allow for this incorporation at a later date. Due to the number of unknown factors, the simulation of actual weapons used in the SE 3015 Robot Competition will be student dependent but a simple analysis of the projectile's flight path and behavior should allow students to tailor their simbot's weapons to reflect their SE 3015 robot weapons.



IX. TARGETING AND TACTICS

Target acquisition and a working weapons system are essential to the robots ability to fight. Effective targeting is dependent on both the optical systems and the student programmed robot code. The robot must be able to obtain a sense of how far away its opponent is and where it is relative to its own orientation in order to attack. Using a projectile type weapon with a limited magazine under the competition time constraints, the robot must be able to make each shot as accurate as possible.

A. TARGET ACQUISITION

The both the SE 3015 robot and the Simbot rely solely on its returned eye values to determine the relative location of its opponent on the playing field. The robot's program helps it determine the speed and direction of its approach to the opponent. The left and right discrimination aids the robot in finding the bearing to opponent.

1. Initial Acquisition and Search Routines

At the start of the competition the SE 3015 robots are placed in opposite corners at one end of the playing field with a divider, see Figure 3.1, between them to initially block their peripheral view. The robots must move past this divider and make their initial target acquisition. This is easily done by simply instructing a robot to move forward past the divider and then begin looking while slowing spinning on axis until it acquires the opponent. Students in the past (with different initial conditions) have used other more complicated routines such as "sprinting" to the far side of the playing field or developing an elaborate routine to determine

which side of the playing field that it is on, then moving forward and turn in the appropriate direction to acquire its opponent.

After the robot has gotten past its initial acquisition phase, it will begin its targeting routine on its acquired target. Although, the robots can generally see the maximum diagonal length of the playing field (approximately 22 feet) it can lose sight of its opponent. A routine to reacquire a lost target must be include in the robot's code. This search routine can be as simple as slowly spinning on axis until it reacquires the target.

The Simbot will respond similarly to its SE 3015 counterpart, except the simulation playing field currently lacks the divider. Without this divider, the Simbot will be able to automatically see its opponent if the robot code instructs immediate search. This omission may cause the Simbot to behave differently than the SE 3015 robot on the actual playing field (see Section C) concerning obstacles . The simulation relies on the student having developed a robot code that has the robot move forward past the divider before it begins searching.

2. Information Processing and Code Optimization

As a general rule for the Robot Wars competition, the robot that can find and accurately shoot its opponent on a consistent basis will win the competition. In order to achieve this, the student needs to program the robot to quickly and accurately evaluate the signal levels of the robot eyes. Inherent to circuitry of the robots are the problems of background noise, interference and signal discrimination which may cause faulty eye readings that must be overcome (see Chapter VI for more discussion on the optical system). The student is seeking to find the clearest signal, either from the circuits or from processing the signals from the eyes.

The robot code can be designed to obtain the cleanest signal levels from the eyes and filter out any anomalies caused by faulty readings using various algorithms before passing the information onto the rest of the code to evaluate for targeting. Previous strategies to eliminate the errors have ranged from ignoring them to elaborate schemes involving normalization and statistical averaging. Ignoring this problem can cost the robot time by acting on a faulty reading and possibly losing sight of the opponent robot. Computationally intense code that weeds out the error before the robot moves can cost time while the computer makes its decision.

3. Decoy Lights and Background Lights

The SE 3015 competition rules change from quarter to quarter, but in general decoy lights of one sort or another have been allowed. As a rule, the decoy lights are not at the same frequency or intensity of the robot beacon. Reflection of the light off the floors, walls, obstacles and the other robot can also cause faulty signal levels.

Decoy lights can be deployed from the robot or remain attached to its body. A more intense signal at a frequency close to the beacon, at a harmonic frequency of the beacon, can saturate the opponents optics and make the robot think it is seeing the opponent at the wrong position. If the robot thinks its opponent is closer than what it really is, the robot may fire inaccurately, subsequently missing and using up limited ammunition.

The simulation currently does not allow for additional light sources on the playing field nor does it incorporate the effects of background lights and reflectivity.

B. TACTICS

In general, the robot will use the eye signal levels in a case statement to determine its next move. If the opponent is too far, the robot will move closer, if the opponent is on the left the robot will move to the left, and if the opponent is nowhere in sight, the robot will use a search routine until it detects it. A sample targeting flow is provided in Figure 9.1. Figure 9.2 shows how the routine would be written in Dynamic C for the robot and Figure 9.3 shows how this sample routine would be written in C++ for the Simbot. This basic decision tree allows the robot to find its opponent, narrow in on it, and then fire.

```
1. get signal levels for left and right eyes
2. sum left and right eyes

3. if  sum too small (i.e. not in firing range)
        move closer
   else if sum too large (i.e. saturation)
        move away
   else if left > right
        move to right
   else if right > left
        move to left
   else if  left ≈ right and close to opponent
        fire weapon
   else (background level)

4. move robot accordingly

5. if time and ammo allow
        repeat steps
```

Figure 9.1 Generic targeting flow.

```
float left, right, sum;
left = ad_rd8(0);
right = ad_rd8(1);
sum = left + right;

while (ammo > 0) {
   if  ((sum < 400) && (sum > 16))
         strcpy(buf, "ffgrffgr");
   else if (sum > 2000)
         strcpy(buf, "frgrfrgr");
   else if (left > right) && ((left - right) > 50)
         strcpy(buf, "ffgrfrgr");
   else if (right > left) && ((right - left) > 50)
         strcpy(buf, "frgrffgr");
    else if  (fabs(left - right) <= 50) && (sum > 800))
         outport(PIODB,xx01);
    else
         strcpy(buf, "ffgrfrgr");

   platform( );
   for (count = 0; count < 20; count ++);

} /* end of while loop */
```

Figure 9.2 Targeting flow (Figure 9.1) written in Dynamic C.

```
float left, right, sum;
left = botA.ad_rd8(0, botB);
right = botA.ad_rd8(1, botB);
sum = left + right;

while (1) {
   if  ((sum < 400) && (sum > 16))
         strcpy(botA.buf, "ffgrffgr");
   else if (sum > 2000)
         strcpy(botA.buf, "frgrfrgr");
   else if (left > right) && ((left - right) > 50)
         strcpy(botA.buf, "ffgrfrgr");
   else if (right > left) && ((right - left) > 50)
         strcpy(botA.buf, "frgrffgr");
   else if  (fabs(left - right) <= 50) && (sum > 800))
         botA.outport(PIODB,xx01);
   else
         strcpy(botA.buf, "ffgrfrgr");

   for (botA.count = 0; botA.count < 20; botA.count++)
        platformA( );
}
```

Figure 9.3 Targeting flow (Figure 9.1) written in C++ for Simbot botA.

The basic tactic shown in Figure 9.1 is slow and cumbersome. The robot will spin on its axis until the observed signal level is above background, if the difference between the left and right eye signal levels is great, it will continue spinning until the difference between levels is below an arbitrary set value. After the robot has decide that it is looking directly at its opponent, the robot will move forward to get within shooting range. As it is moving, if the difference in eye signal levels indicates that it is no longer pointed directly at the opponent, the robot will stop and begin turning until it is on target once again. Once the robot is close enough it will fire. An overhead view of the playing field would show the robot moving in a zig zag fashion.

In order to defeat the opponent, the robot may have to use more complicated tactics to acquire the target and shoot. More elaborate targeting schemes breaks down the signal levels to reflect approximate distances from the opponent. At farther distances, the robot may move faster than at shorter distances. The student may incorporate curves and corner turns into the targeting routines to line the robot up for the shot. At the point of firing, students have used different tactics on how much to shoot when.

The easiest tactic for attacking the opponent is to shoot all the rounds in the magazine once the robot is in range. The fault in this tactic is that once the robot is in firing range it is also generally in range to be fired upon by its opponent's. Other tactics have included shooting a few rounds, moving out of range of the opponents weapon and then reacquiring the opponent, repeating this until it has no more ammunition. With larger magazines, the robots may not need to go to a defensive routine since the robot may be able to continuously shoot at its opponent for

the duration of the competition. For smaller magazines, the student may have to include defensive tactics to keep out of range of its opponent which may still have ammunition.

The nature of the competition will dictate the need for tactics. With a small playing field and unlimited magazines for the weapons, it is possible for the competition be nothing more than a war of attrition. The robot with the larger magazine (and functioning weapon) will be the victor. If the playing field is large enough that the optical system cannot accurately track its opponent across the playing field and the weapons range also cannot reach the limits of the field, tactics and target acquisition become more critical.

C. DEPLOYED OBJECTS, OBSTACLES, MINES, AND DECOYS

Previous SE 3015 Robot Wars Competitions have allowed obstacles on the playing field. These obstacles have include such things as wood blocks, decoy screens, and light decoys. The obstacles are generally deployed from a robot to impair the movement of its opponent (and sometimes itself). In order to simulate these objects, the simulation architecture will have to keep track of each of these items as it tracks the two Simbots. Incorporation of these obstacles will require adding a more detailed collision detection method in the simulation robot class and a greater knowledge of the obstacle itself.

The easiest type of obstacle to incorporate into the simulation is a rigid body that does not move when a Simbot collides with it on the playing field. This type of obstacle has been used in the SE 3015 competition in the past in the form of a weighted mine placed on the playing field by the instructor. Obstacles that are placed on the playing field by deployment off a robot are of a much lighter weight and may or may not move when impacted by a robot.

In the March 1996 SE 3015 Robot Wars Competition, students deployed wooden blocks from their robot onto the playing field to impede the motion of the opponent. The blocks were wrapped in non skid material to keep them from sliding across the floor. The goal was for the SE 3015 robot to run into the blocks and activate the bumper routine. In some cases this worked, in others, the block did not appear to even slow down the robot.

Modeling the interaction between the robots and the obstacles in the simulation would require knowing the weight, dimensions, coefficient of friction and other such details explicitly for each of the obstacles. The playing field divider, also an obstacle, has not been incorporated into the simulation for this reason. The problem of simulating obstacles has been left for future research.

D. SUMMARY

The robot wars competitions vary from year to year in the use of weapons, presence of obstacle and size of the playing field, but the basic requirement to find and shoot the opponent remains. Quick and accurate target acquisition, a large magazine and a working weapon are the key components to victory. The simulation currently incorporates these basic principles and allows the student opportunity to test their target acquisition strategy on an open playing field.

X. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

The simulation of the SE 3015 Robot Wars competition was created to extend the students' understanding of weapon system design to the use of computer simulation and modeling. This simple model of the competition provides the ability to implement an individual's robot code into a simulated environment. The simulation does not provide a true physical environment rather a synthetic environment in which to test robot design and code against an opponent.

This simulation foundation of the Simbot and playing field environment is an ideal computer version of the SE 3015 Robot Wars Competition ignoring most of the real world performance parameters. The following are the capabilities included: integration of the Dynamic C robot code, left and right discrimination for targeting, optical sensor saturation limits, projectile velocity and basic robot motion. For a listing of the pertinent physical parameters and which one are included in the simulation, see Chapter III, Section B.

The simulation is able to use the SE 3015 robot code for the Simbot without loss of original logic and tactics programmed by the user. The simulation does require, that the user provide some modification the function names and code lines for use in the simulation, but these have been kept to a minimum. The simulation contains the basic movement functions of the robots. It is able to interpret the motor control string commands accurately and move accordingly in the simulation environment graphical display. The optical portion of the simulation accurately handles left and right discrimination in the eyes to determine the relative angle of the opponent on the playing field. The simulation can respond to the signal levels

received and move according to the information as dictated by its code. Although the Simbot will respond correctly to the received signal levels, it lacks the ability to provide a true signal level for each eye commensurate with the relative distance of the opponent. The weapons portion uses a simple projectile motion to fire at the opponent when the simCode determines a targeting solution has been achieved. If the Simbot is close enough and at the correct angle of fire, the projectile will hit its target, else it will miss. The weapons lack the dispersion and errors found in the actual weapons.

"Simulations are doomed to succeed," (often quoted, source unknown) and this one is no exception. The observer of the simulation sees reasonable looking play with no indication of whether or not it is realistic. Lacking the randomness of the physical world, simulation fails to illustrate the environmental conditions that have cost competitor's their victory in the past. Dead batteries, fried circuit boards, jammed weapons, etc., have not been integrated into the simulation. The current simulation only employs the most basic of physics in order to get the simulation started but needs to incorporate a larger number of concepts.

A test of the Dynamic C code in Appendix B was successfully integrated into the simulation as per the instructions in Appendix E. The Simbot, however, did not perform as expected. The simulation revealed a flaw in the original Dynamic C code that was not previously discovered. The code is missing a case for when only one eye detects the opponent and the other eye only detects the background signal level. During the Robot Wars competition, where the robot never lost sight of its opponent and the robot performed correctly. In the simulation the Simbot lost sight of its opponent in one eye and was not able to continue to function, requiring the simulation to be abnormally terminated. This incident illustrates the

potential value of the simulation to detect logic errors within the tactics since it can test for more cases. Appendix G, contains a video of the simulation demonstrations prepared to show the logic test capabilities.

B. RECOMMENDATIONS FOR FUTURE WORK

Future work should include a true optics simulation that incorporates the signal loss due to spreading and the efficiencies of the photo diodes and detectors. The robot base needs to incorporate calibrated angular velocities that correctly replicate the SE 3015 robot actions. The standard weapon needs to be extended beyond a simple projectile calculation to improve its functionality in the simulation. The Simbot needs to be able to handle multiple types of weapons. The interrupt functions need to be incorporated into the simulation of each of the Simbots vice the generic interrupt found in the Simbot class.

The simulation needs to incorporate a Graphical User Interface (GUI) to allow the Simbots to be tailored to the specific robots modeled. Additional set functions need to be added to the Simbot class to fine tune the individual Simbots to their physical counterpart. A translator program which will take the Dynamic C code for transparent conversion of Dynamic C code into the simulation is needed. A better graphics display needs to be added, possible one to allow internet display of the competition.

C. SUMMARY

The simulation foundation incorporates a class structure to model the robot, C++ simulation code to allow the playing field interactions and graphics code to display the simulated

Robot Wars Competition. The fundamental physical movement of the robot has been correctly modeled and verified. The basic goal of the thesis has been accomplished. A simulation structure has been created that includes the basics of the Robot Wars and that provides visualization of the competition. The simulation has been designed so that the physical fidelity can be improved in a modular fashion.

APPENDIX A. ROBOT WARS COMPETITION RULES

The most current rules for the SE 3015 Robot Wars Competition (Harkins, 1997) is provided. Whenever possible, the simulation should comply with the rules of the current course taught. The rules for the Robot Wars Competition are not static and can vary widely from one course to the next. The most rule changes involve the use of weapons and decoys. The course taught in Winter Quarter 1997, did not allow for decoys or included mines but allowed additional weapons. The course taught the previous year allowed for the mines and decoys but additional weapons, other than the standard gun were not permitted.

Robot Construction and Competition Rules

Robot Requirements and Restrictions:

1. Height and weight:

Maximum weight limit: 35 lb..
Maximum height limit: 4.0 ft.

2. Beacon frequency assignments:

These have been issued and are set by a trim pot on the beacon oscillator board. Each team has been assigned a unique frequency.

The beacon will be mounted in a centered position underneath the top "deck" of the robot platform as discussed in class. Each beacon will give an unobstructed 360 degree view to the opponent (platform posts and the occasional wire is obviously accepted).

3. Weapons:

A standard weapon will be issued to each team.

A secondary weapon system is encouraged but not required.

The standard weapon is sufficient for final competition. The choice to not employ a secondary weapon system will not be held against the team for competition or grade purposes.

3. Magazine limitations.

The size of the standard gun magazine will remain as issued. Modifications to this magazine is not authorized.

The size and scope of the secondary weapon system, if employed, will be limited by the height and weight limitations already listed for the platform. The magazine for the second weapon is limited to fifteen rounds.

4. Countermeasures:

Countermeasures are authorized at the discretion of the instructor. If you have an idea check with me prior to use.

5. Competition code:

The robot program code will be fixed at the time of competition. This is to encourage pre competition testing.

Code modifications between rounds will not be authorized. You will have to " Dance With The One You Brung".

You will be allowed to reload if the system crashes.

6. General competition rules:

The final event for the course will be a round robin *competition with the teams being seeded in the opening brackets in a random fashion.*

Each competition round will last three minutes.

Your platform must demonstrate the classic "Detect to Engage" capability. This means you must have distinct and obvious search, detect and engage functions within your code. Shooting in the "shotgun" or "scatter" mode without regard to where your opponent is not allowed.

A valid hit is when your weapon or projectile strikes your opponent. Strikes off of floor skips are valid as long as they are not a ricochet off of some other device, for example a chair, a leg or the mountain. Assigned judges will determine the number of hits per competition. The platform that inflicts the most valid hits on the opponent in three minutes will be declared the winner by the judges. The judges may walk on the playing field during the competition to ensure accurate counting.

Your weapon may not inflict actual damage upon your opponents platform. I reserve the right to veto the employment of any device that is dangerous.

Ties, though not common, will be adjudicated by the flip of a coin. This is not very common but effective.

On occasion there is a round of competition that simply needs to be restarted for some reason. I reserve the right to do this as required.

APPENDIX B. SAMPLE SE 3015 DYNAMIC C ROBOT CODE

The Dynamic C code attached was provided by Wm B. McNeal and Jerry Sullivan and was used in the Winter Quarter 1997 Robot Wars Competition. The code is written with while loops and incorporates five functions: fire, search, and read in addition to the required functions: interrupt and platform. The fire function turns on the weapons and fires the rounds. The search function is used when the robot lost detection of the opponent and provides a serious of manuvers designs to locate the opponent. The read function obtains the eye signal levels from the A/D converter.

```
/*****************************************************************
*****************************************************************
*****************************************************************
LT WM. B. McNEAL/LT JERRY SULLIVAN
SE 3015
COMPUTER PROGRAM FOR ROBOT COMPETITION  (ICEMAN)
*****************************************************************
*****************************************************************/
#use"serial.lib"
#INT_VEC PIOA_VEC INT1
char buf[9];
char cc;
int x,y,i,cnt, s, shcnt;
unsigned long int count;
main()
{
ser_init_z1(0,9600/1200);
hv_enb();
outport(PIOCA,0xff);
outport(PIOCA,0xff);
outport(PIOCA,PIOA_VEC);
outport(PIOCA,0x97);
outport(PIOCA,0xfc);
outport(PIOCA,0x87);
outport(PIOCB,0xff);
outport(PIOCB,0x00);

cnt=0; /*count for lost tracking if target is lost for approx. 10 secs search routine
starts again*/

shcnt=0; /*this counter counts number of times eye detection criteria makes robot go
straight
          as long as robot goes straight conter counts, if robot turns count goes to
zero
          this ensures robot has truly acquired target before it shoots*/
read();  /* read for initial search routine */

for(count=0;count<50000;count++); /* this count is delay for search routine to let
opponent get out first*/
search(); /* search routine */

while(1){
        if (shcnt==17){/*if shoot count gets to 17 target has been successfully
acquired*/
         fire();
         fire();
         fire();
         shcnt=0;
         }
         read();
/*printf ("\nCH0 = %d     CH1 = %d",x,y);*/


/* THIS PORTION OF THE PROGRAM TAKES THE VALUES FOR THE LEFT AND RIGHT EYES COMPARES
THEM AND EITHER
   TURNS LEFT, TURNS RIGHT OR GOES STRAIGHT */


/* SLOW SPIN SEARCH */
  if (x<=48 && y<=48){
    strcpy(buf,"4fgr4rgr");
    platform();
    cnt++; /* count for lost detection */
    shcnt=0;
```

```
    /*printf("\ncount = %d",cnt);*/
    if (cnt==500){ /* if lost detection count gets to 500 go back to search routine */
      cnt = 0;
      search();}
                    }

  if (x>48 && y>48){

/* FINAL DETECTION MODE  stand off criteria*/
    if ((x>=400 && y>=400)  && (abs(x-y)<=50)){
    cnt=0;
    shcnt++;
    strcpy(buf,"8fsr8fsr");  /* stand off */
    platform();
    for(count=0;count<1000;count++);
    fire();
    fire();
                                          }

/* BOTH EYES EQUAL NOTE QUITE THERE */
    if (((x<400 && x>=250) && (y<400 && y>=250))  && (abs(x-y)<=50)){
    cnt=0;
    shcnt++;
    strcpy(buf,"afgrafgr");
    platform();
                                                      }

/* BOTH EYES EQUAL STILL  APPROACHING */
    if (((x<250 && x>=100) && (y<250 && y>=100))  && (abs(x-y)<=50)){
    cnt=0;
    shcnt++;
    strcpy(buf,"cfgrcfgr");
    platform();
                                                          }

/* BOTH EYES EQUAL INITIAL ACQUISITION */
    if ((x<100 && y<100)  && (abs(x-y)<=50)){
    cnt=0;
    shcnt++;
    strcpy(buf,"ffgrffgr");
    platform();
                                          }

/* SHARP LEFT TURN DIFFERENCE>2500 */
    if (x-y>2500){
    cnt=0;
    shcnt=0;
    strcpy(buf,"9fsrcfgr");
    platform();
    for(count=0;count<1000;count++);
                }

/* SLOWER SHARP LEFT TURN DIFFERENCE>1000 BUT LESS THAN 2500 */
    if (x-y>1000 && x-y<=2500){
    cnt=0;
    shcnt=0;
    strcpy(buf,"9fsrefgr");
    platform();
    for(count=0;count<1000;count++);
                              }

/* SLOW LEFT TURN DIFFERENCE >500 BUT LESS THAN 1000*/
    if (x-y>500 && x-y<=1000){
    cnt=0;
```

```
    shcnt=0;
    strcpy(buf,"argrafgr");
    platform();
    for(count=0;count<1000;count++);
                              }

/* LEFT TURN DIFFERENCE >50 BUT LESS THAN 500*/
    if (x-y>50 && x-y<=500){
    cnt=0;
    shcnt=0;
    strcpy(buf,"8rgr8fgr");
    platform();
    for(count=0;count<1000;count++);
                             }

/* SHARP RIGHT TURN DIFFERENCE>2500 */
    if (y-x>2500){
    cnt=0;
    shcnt=0;
    strcpy(buf,"cfgr9fsr");
    platform();
    for(count=0;count<1000;count++);
                }

/* SLOWER SHARP RIGHT TURN DIFFERENCE>1000 BUT LESS THAN 2500*/
    if (y-x>1000 && y-x<=2500){
    cnt=0;
    shcnt=0;
    strcpy(buf,"efgr9fsr");
    platform();
    for(count=0;count<1000;count++);
                              }

/* SLOW RIGHT TURN DIFFERENCE>500 BUT LESS THAN 1000*/
    if (y-x>500 && y-x<=1000){
    cnt=0;
    shcnt=0;
    strcpy(buf,"afgrargr");
    platform();
    for(count=0;count<1000;count++);
                              }

/* RIGHT SEARCH TURN DIFFERENCE>250 BUT LESS THAN 500*/
    if (y-x>50 && y-x<=500){
    cnt=0;
    shcnt=0;
    strcpy(buf,"8fgr8rgr");
    platform();
    for(count=0;count<1000;count++);
                              }
                  }
        }

}
interrupt reti INT1() /* INTERRUPT ROUTINE */
{
int data;
unsigned long int counter;
EI();
data=inport(PIODA);
if(data==253)
  {
  strcpy(buf,"crgrcrgr"); /* IF FWD INT IS TRIPPED GOES BACKWARDS */
  platform();
```

```
  for(counter=0;counter<7500;counter++);
  strcpy(buf,"argrafgr");  /* AFTER BCKWRDS THEN SPINS TO FREE ITSELF */
  platform();
  for(counter=0;counter<5000;counter++);
  }
if(data==254)
  {
  strcpy(buf,"cfgrcfgr"); /* IF AFT INT IS TRIPPED GOES FORWARD */
  platform();
  for(counter=0;counter<7500;counter++);
  strcpy(buf,"afgrargr"); /* AFTER FWD THEN SPINS TO FREE ITSELF */
  platform();
  for(counter=0;counter<5000;counter++);
  }
return;
}


platform()
{
int z;
cc = sizeof(buf);
outport(ENB485,1);
ser_send_z1(buf,&cc);
for(z=0;z<700;z++);
}


search() /* ICEMAN SEARCHES INITIALLY AND IF CONTACT IS LOST FOR MORE THAN 10 SECS */
{
unsigned long int counter;
int m;
 if (x<=48 && y<=48){ /*FIRST ICEMAN GOES STRAIGHT*/
   strcpy(buf,"cfgrcfgr");
   platform();
     for (counter=0;counter <=8000;counter++){
     read();
     if (x>48 && y>48) /* IF AT ANY TIME OPPONENT IS DETECTED PROGRAM GOES BACK TO
MAIN SECTION*/
       return; }
   strcpy(buf,"8fgr8rgr"); /*NEXT ICEMAN SPINS RIGHT*/
   platform();
     for (counter=0;counter <=10000;counter++){
     read();
     if (x>48 && y>48)
       return;}
   strcpy(buf,"5fgrcfgr");  /*THEN ICEMAN SNAKES LEFT*/
   platform();
     for (counter=0;counter <= 5000;counter++){
     read();
     if (x>48 && y>48)
       return;}
       strcpy(buf,"cfgr5fgr"); /* SNAKES RIGHT */
   platform();
     for (counter=0;counter <=5000;counter++){
     read();
     if (x>48 && y>48)
       return;}
   strcpy(buf,"8rgr8fgr"); /* THEN ICEMAN SPINS LEFT */
   platform();
     for (counter=0;counter <=10000;counter++){
     read();
     if (x>48 && y>48)
       return;}
```

```
        }
    return;
  }


fire() /* ROUTINE TO FIRE GUNS*/
 {unsigned long int counter;
 for(counter=0;counter<5000;counter++);
  outport(PIODB,0x40); /*CLOSES SOLENOID FOR DISK SHOOTER*/
  for(i=0;i<10000;i++);
  outport(PIODB,0x88); /*STARTS BOTH GUN MOTORS*/
  for(i=0;i<10000;i++);
  outport(PIODB,0x30);  /*CLOSES SOLENOIDS FOR BOTH GUNS*/
  for(i=0;i<2500;i++);
  outport(PIODB,0x00);  /*SHUTS OFF EVERYTHING*/
  for(i=0;i<5000;i++);
 }


 read() /*READS  LEFT AND RIGHT EYE VALUES FOR COMPARISON IN MAIN SECTION*/
 {
 x = ad_rd8(0);
y = ad_rd8(1);
return;
}
```

APPENDIX C. SIMBOT CLASS

The Simbot Class code consists of a header file, SimbotClass.h and a class file, SimbotClass.C. As this class is only a foundation and will be rewritten to include more detailed models of robot components, the decision has been made to leave all class data members and functions are all public.

The **SimbotClass.h** contains the list of parameters include and there variable names. The function prototypes are listed without description. The descriptions are found in the **SimbotClass.C** file.

In **SimbotClass.C**, only two **set** functions have been developed, **set** to initially position the Simbot on the playing field, and set the magazine size, and **setEyes** to set the geometry of the optical circuit, saturation value and saturation minimum distance. Additional set function can be added to allow for setting more parameters of the Simbot and further differentiating the opponents from each other. Each function has a description preceding it.

Two auxiliary functions, **rad2deg**, and **deg2rad** have been added to allow for easy conversion from radains to degrees and back. The C++ code uses radians for the trigonometric functions but the graphics package used requires degrees.

```
//-------------------------------------------------------------
//FILENAME: SimbotClass.h
//AURTHOR: Doreen M. Jones
//DATE: 31 MAY 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY: This is the header file for Simbot.h.  The class does
// not have any designated private data variables to facilitate future
// development of the class but future research.
//-------------------------------------------------------------

#ifndef SimbotClass_h
#define SimbotClass_h

class SimbotClass {
public:

   float bullets[15][8];
   float guns[3][5];
   float xpos;
   float zpos;
   float theta;
   float theta2;
   float oldang;
   int forward;     // flag to indicate Simbot direction
   int move;        // flag to indicate the Simbot is in motion
   int bumper;      // flag for bumper hit
   int bumper_count;
   int edge;        // flag for edge hit
   int edge_count;
   int count;
   float tick;      // speed factor, make sim run faster or slower
   int timer;

   char buf[8];     // motor control string from program

   int score;
   int loop;
   int edger;

   float floor;

   int color;
   int fire;
   int shoot;

   float left;
   float right;
   float L_width, R_width;
   float nose;
   float max_level, background;
   float MAX_EYE;
   float sat_dist;
   float L_radius, R_radius;
   float separate;

   SimbotClass();
   void set(float xorigin, float zorigin, int mag,
            int paint, float time, float floorSize, int moveable);
   void setEyes(float l_wide, float r_wide, float noze, float highest,
                     float dist, float back);
   int ad_rd8(int nn, SimbotClass& Simbot);
   void outport(int, int);
   void print();
```

```
//---------------------------------------------------------------

   void initialize_gun();
   void optics(SimbotClass& Simbot);
   void edge_detect(int &w1, int &w2);
   void bumpers(SimbotClass& bot2, int &w1, int &w2 );
   void auto_ref(SimbotClass& bot2 );
   void getVelocity(int &w1, int &w2);
   void platform(SimbotClass& bot2);
   void position(int, int);
   void reposition();
   void gunfire();
   void projectiles(SimbotClass& botB);
   float relative_angle(SimbotClass& bot2);
   float range(SimbotClass& bot2);

};

#endif

//---------------------------------------------------------------
// end of file SimbotClass.h
//---------------------------------------------------------------
```

```
//-------------------------------------------------------------
//FILENAME: SimbotClass.C
//AURTHOR: Doreen M. Jones
//DATE: 31 MAY 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY: This file contains the class functions for the Simbot
// class.  Two auxillary functions are included to assist with the
// degree to radian conversions.
//-------------------------------------------------------------

#include <iostream.h>
#include <math.h>
#include "SimbotClass.h"

#include <string.h>
#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>

//-----------------------------------------------------------------------
// auxillary functions
//-----------------------------------------------------------------------
float deg2rad (float degree);
float rad2deg (float radian);
//-----------------------------------------------------------------------



//-----------------------------------------------------------------------
//function: constuctor
//description: creates the basic Simbot object with the Simbot placed at the
// origin and the variables set to a neutral value.  The position of the object
// and other parameters are set in the set and setEyes functions.
//-----------------------------------------------------------------------

SimbotClass::SimbotClass()
{
   xpos = 0;
   zpos = 0;
   tick = 1;
   timer = 0;
   theta = 0; // point the gun up the z-axis.
   forward = 1;
   move = -1;
   edge = 0;
   edge_count = 0;
   bumper = 0;
   bumper_count = 0;
   loop = 0;
   edger = 0;
   shoot = 0;
   initialize_gun();
   guns[0][0] = 14.0;
   guns[1][0] = 14;
   guns[0][2] = 1050.0;
   guns[1][2] = 1050.0;
   guns[0][1] = 14;
}
```

```
//---------------------------------------------------------------------------
//function: set
//description: similar to the constructor put is specifically for setting the
// initial position of the Simbot, the weapon parameters and the color of the
// simbot for use by the graphics program.
//---------------------------------------------------------------------------
void SimbotClass::set(float xorigin, float zorigin, int mag, int paint,
          float time,  float floorSize, int moveable)
{
   xpos = xorigin;
   zpos = zorigin;
   color = paint;
   tick = time ;
   floor = floorSize;
   move = moveable;
   fire = 0;
   left = 0;
   right = 0;
   guns[0][0] = mag;
   guns[1][0] = 14;
   guns[0][2] = 1050.0;
   guns[1][2] = 1050.0;
   guns[0][1] = 14;
}



//---------------------------------------------------------------------------
// function: setEyes
// description: this function will set the values for the eye geometry, the
//  maximum eye level, background and saturation distance.  It uses the last
//  three to calculate the Kappa to determine the eye signal levels at the
//  ranges from the Simbot to its opponent.
//---------------------------------------------------------------------------
void SimbotClass::setEyes(float l_wide, float r_wide, float noze, float highest,
                          float dist, float back)
{
   L_width = l_wide;
   R_width = r_wide;
   nose = noze;
   max_level = highest ;
   sat_dist = dist;
   background = back;
   MAX_EYE = ((max_level - background)*sat_dist*sat_dist);
}


//---------------------------------------------------------------------------
// function: ad_rd8
// description: interface function used by the robot to obtain the eye signal
//  levels detected by the sensors.
//---------------------------------------------------------------------------
int SimbotClass::ad_rd8(int nn, SimbotClass &botB)
{
   optics(botB);
   if (nn == 1) {
      return right;
   }
   else {
      return left;
   }
}
```

```
//---------------------------------------------------------------------------
// function: optics
// description: computes the eye signal levels for each of the eyes based on
//   the range between the two Simbots, then if in the sights of the Simbot, the
//   eye signal values will be calculated using Formula 7.2, the levels for each
//   eye are determined by Formula 7.3 and 7.4
//---------------------------------------------------------------------------


void SimbotClass::optics(SimbotClass &botB)
{
   float dist = range(botB)/100.0;
   float ang = relative_angle(botB);
   float diff = theta - ang;
   float adif = fabs(diff);
   float percent = 0 ;
   float eye, phi;
   int side;
   const int RIGHT = 0, LEFT = 1, BOTH = -1;

   if ((adif < 90) || (adif > 270)) {  // in sites
      if (diff == 0.0 || diff == 360.0) {
         side = BOTH;
      }
      else if ((diff < 90.0 && diff > 0.0) || (diff  < -270.0)) {
         side = LEFT;
      }
      else if ((diff > -90.0 && diff < 0.0) || (diff > 270.0)){
         side  = RIGHT;
      }
   }

   else { // it doesn't see anything except background, so exit
      right = left = 16.0;
      return;
   }

   if (side == RIGHT) {  // left side is obscured
      phi = rad2deg(atan(L_width/nose));
   }
   else if (side == LEFT) {  //right side is obscured
      phi = rad2deg(atan(R_width/nose));
   }

   if ((adif < phi) || (adif > (360.0-phi))) {
      if (adif > 270.0) {
         percent = 1.0 - ((360.0 - adif)/phi);
      }
      else {
         percent = 1.0 - (adif/phi);
      }
   }
   else {
      percent = 0.0;
   }

   if (dist > sat_dist) {
      eye = (MAX_EYE)/(dist*dist) + background;
   }
   else {
      eye = max_level;
   }

   if (side == LEFT) {
```

```
      right = eye;
      left = eye * percent;
      if (percent == 0.0) {
         left = background;
      }
   }
   else if (side == RIGHT) {
      right = eye * percent;
      left = eye;
      if (percent == 0.0) {
         right = background;
      }
   }
   else {
      left = right = eye;
   }

}


//-------------------------------------------------------------------------
// function: edge_detect
// description: determines if the edge has been crossed by the Simbot and then
//   correct it so that the Simbot will not exit the playing field.  If the edge
//   has been detected, the Simbot will act on the movement sequence in this
//   function before finishing the rest of its movement routine from the main
//   program.
//-------------------------------------------------------------------------

void SimbotClass::edge_detect(int &w1, int &w2)
{
   float edge_pt = 55; // this is the location of the edge detector from Simbot axis
   float xtemp = edge_pt*sin(theta2);
   float ztemp = edge_pt*cos(theta2);
   float forx, forz, aftx, aftz;
   forx = xtemp + xpos;
   forz = ztemp + zpos;
   aftx = xpos - xtemp;
   aftz = zpos - ztemp;

   if (fabs(forx) > floor) {
      edge = 1; // forward edge
      edger++;
      edge_count = 40;
   }
   if (fabs(forz) > floor) {
      edge = 1; // forward edge
      edger++;
      edge_count = 40;
   }
   if (fabs(aftz) > floor) {
      edge = 2; // back edge
      edger++;
      edge_count = 40;
   }
   if (fabs(aftz) > floor) {
      edge = 2; // back edge
      edger++;
      edge_count = 40;
   }
   if (edge_count <= 0) {
      edge = 0; // no edge detected.
      reposition();
      edger = 0;
```

```
    }
    else {  // finish existing edge detection routine
       if (edge == 1) {
          if (edge_count > 30) {
              w1 = -14; w2 = -15;
          }
        else {
             w1 = 0; w2 = 15;
          }
       }
       else if (edge == 2) {
          w1 = 15; w2 = 14;
       }
       else {
          return;   // escape there should be any other values but 1 or 2
       }
       edge_count--;
    }

}


//-------------------------------------------------------------------------
// function: bumpers
// description: this checks to see if the Simbot has bumped into anything and
// then responds appropriately if it has.  This reaction takes precedence over
// the movement and over the edge detection.  This program checks for multiple
// occurance of the routine that have failed to finish.
//-------------------------------------------------------------------------

void SimbotClass::bumpers(SimbotClass& bot2, int &w1, int &w2)
{
   float radiusX2 = 120.0;
   float temp = range(bot2);

   if (temp <= radiusX2) {
      if (forward == 1) {
           bumper = 1;
           loop++;
      }
      else {
           bumper = 2;
           loop++;
      }
      if ((bot2.theta > (theta+90.0)) && (bot2.theta > (theta-90.0))) {
          bot2.bumper = 2;
          loop++;}
      else {
          bot2.bumper = 1;
          loop++;}

      bot2.bumper_count = 40;
      bumper_count = 40;
   }
   else if (bumper_count == 0){
      bumper = 0;
      reposition();
      loop = 0;
   }

   if (bumper == 1 && bumper_count > 0) {
      w1 = w2 = -15;
      bumper_count--;
   }
```

```
   else if (bumper == 2 && bumper_count > 0) {
      w1 = w2 = 15;
      bumper_count--;
   }
   auto_ref(bot2);

   return;
}


//---------------------------------------------------------------------------
// function: auto_ref
// description: this function attempts to correct for the cases where there are
// multiple unfinished edge or bumper routines.  The function corrects by trying
// to rotate the robots or move them aside to get them out of the problem.
//---------------------------------------------------------------------------

void SimbotClass::auto_ref(SimbotClass& bot2)
{
   if (loop > 5 && edger > 5) {
     xpos += 50;
     zpos += 50;
     bot2.xpos += 50;
     bot2.zpos += 50;
        loop = edger = 0;
   }
   else if (loop > 5) {
      theta += 45.0;
      bot2.theta += 45.0;
      loop = 0;
   }
   return;
}


//---------------------------------------------------------------------------
// function: position
// description: calculates the next position and rotation of the Simbot on
// the playing field.
//---------------------------------------------------------------------------

void SimbotClass::position(int w1, int w2)
{

   const float r = .25, d = 1.0;
   float theta1;
   float v1 = w1 * r;
   float v2 = w2 * r;

   theta1 = (2.0* (v1-v2) / d) * tick;
   theta += theta1;

   // calculate the total velocity
   float first = (v1 + v2) / 2;
   // determine what direction
   if (first >= 0 ) {
     forward = 1;
   }
   else {
     forward = 0;
   }

   theta2 = deg2rad(theta);
```

```
   float tx = first*sin(theta2)*tick;
   float tz = first*cos(theta2)*tick;

   if (theta > 360 || theta < -360) {
      int temp = theta/360;
      theta = theta - float(temp*360);
   }
   if (theta < 0){
      theta += 360;
   }
   if (theta == 0 || theta == 180) {
     tx = 0;
   }
   else if (theta == 90 || theta == 270) {
     tz = 0;
   }

   xpos = xpos + tx;
   zpos = zpos + tz;

}


//-------------------------------------------------------------------------
// function:reposition
// description: This function will check to make sure that the Simbot has not
// moved off the playing field and return it to a point near where it left, if
// it has left.
//-------------------------------------------------------------------------

void SimbotClass::reposition()
{
   if (xpos < -floor) {
     xpos = - 650;
   }
   else if (xpos > floor) {
      xpos = 650;
   }
   else if (zpos < -floor) {
      zpos = -650;
   }
   else if (zpos > floor) {
      zpos = 650;
   }
   return;
}


//-------------------------------------------------------------------------
// function: getVelocity
// description: this function breaks down the motor control string into the
//   appropriate angular velocities for the wheels.  This function assumes that
//   valid motor control strings have been entered.
//-------------------------------------------------------------------------

void SimbotClass::getVelocity(int &w1, int &w2)
{
   w1 = buf[0];
   w2 = buf[4];

   if (w1 > 96) {
      w1 -= 87;
   }
   else {
```

```
      w1 -= 48;
   }

   if (buf[1] == 'r') {
      w1 = -w1;
   }

   if (buf[2] == 's') {
      w1 = 0;
   }

   if (w1 < -15 || w1 > 15) {
      cout << "!!!!!! ERROR !!!!!!" << endl;
      cout << "buffer string " << buf << "is wrong" << endl;
   }

   if (w2 > 96) {
      w2 -= 87;
   }
   else {
      w2 -= 48;
   }

   if (buf[5] == 'r')
      w2 = -w2;
   if (buf[6] == 's')
      w2 = 0;
   if (w2 < -15 || w2 > 15) {
      cout << "!!!!!! ERROR !!!!!!" << endl;
      cout << "buffer string " << buf << "is wrong" << endl;
   }

   return;
}


//---------------------------------------------------------------------------
// function: platform
// description: This function will move the Simbot, and the projectiles by
// calling the appropriate functions.  The order of sequence is to move the
// projectiles, check the bumper, check the edge and then if both or okay then
// move the Simbot.
//---------------------------------------------------------------------------

void SimbotClass::platform(SimbotClass& bot2)
{
   int w1, w2;

   projectiles(bot2);
   bumpers(bot2, w1, w2);

   if (bumper == 0) {
      edge_detect(w1, w2);
      if (edge == 0)
         getVelocity(w1, w2);
      else
         count--;
   }

   position(w1, w2);
}
```

```
//--------------------------------------------------------------------------
// function: initialize_gun
// description: this function initialized the matrix used for the bullets
//--------------------------------------------------------------------------

void SimbotClass::initialize_gun(){

   for ( int ii = 0; ii < 15; ii++) {
      for ( int jj= 0; jj < 8; jj++){
      bullets[ii][jj] = 0.0;
      }
   }

}


//--------------------------------------------------------------------------
// function: gunfire
// description: this function obtains the starting position of each of the
// projectiles at the time from the Simbot's position and gun angle
//--------------------------------------------------------------------------

void SimbotClass::gunfire()
{
   if (guns[0][1] >= 0){
      shoot = 1; // sets a color flag for the graphics
      int mag;
      mag = guns[0][0];
      int round;
      round = guns[0][1];
      guns[0][1] -= 1.0;
      bullets[round][0] = 1; // make the bullet active
      bullets[round][1] = theta2;  // firing angle
      bullets[round][2] = xpos;
      bullets[round][3] = 150.0;// gun height
      bullets[round][4] = zpos;
      bullets[round][5] = guns[0][2]; // projectile velocity
  }
  else {
       shoot = 0;
  }

}


//--------------------------------------------------------------------------
// function: projectiles
// description: this function at each time step will check the bullet matrix
// for active bullets and move them if appropriate.  This function will check
// to see if the bullet has hit the other Simbot or missed.
//--------------------------------------------------------------------------

void SimbotClass::projectiles(SimbotClass& botB)
{
   int mag;
   mag = guns[0][0];

   for (int ii = mag ; ii >= 0; ii--) {
      if (bullets[ii][0] == 1) {
         if ((range(botB)) < (bullets[ii][5]*tick)){
            float secs = ((range(botB)) / (bullets[ii][5]*tick));
            bullets[ii][2] += sin(bullets[ii][1])*bullets[ii][5]*secs;
            bullets[ii][3] += -0.5*980.0*secs*secs;
            bullets[ii][4] += cos(bullets[ii][1])*bullets[ii][5]*secs;
```

```
            if ((bullets[ii][3]) <= 0.0) {
               cout << "missed" << endl;
               bullets[ii][0] = 3;
            }
            float bull = sqrt(((bullets[ii][2]-botB.xpos)*
                              (bullets[ii][2]-botB.xpos))
                              + ((bullets[ii][4]-botB.zpos)*
                                (bullets[ii][4]-botB.zpos)));
            if (bull <= 100.0) {
               cout << "hit Simbot" << endl;
               bullets[ii][0] = 2;
            }
            else  {
               cout << "missed" << endl;
               bullets[ii][0] = 3;
            }

         }
         else {
            bullets[ii][2] += sin(bullets[ii][1])*bullets[ii][5]*tick;
            bullets[ii][3] += -980.0*tick;
            bullets[ii][4] += cos(bullets[ii][1])*bullets[ii][5]*tick;
            if ((bullets[ii][3]) <= 0.0) {
               cout << "hit the ground" << endl;
               bullets[ii][0] = 3;
            }
            if ((fabs(bullets[ii][2])) > floor){
               cout << "missed" << endl;
               bullets[ii][0] = 3;
            }
            if ((fabs(bullets[ii][4])) > floor){
               cout << "missed" << endl;
               bullets[ii][0] = 3;
            }
         }
      }
   }
}


//--------------------------------------------------------------------------
// funciton: print
// description: used to dump the parameters on to the screen if desired.
//--------------------------------------------------------------------------

void SimbotClass::print()
{
     cout << "----- Simbot statistics -----" << endl
          << "xpos = " << xpos << " zpos = " << zpos << endl
          << "theta = " << theta << " buf = " << buf << endl
             << " move = " << move << " count = " << timer << endl
          << "bumber = " << bumper << " count = " << bumper_count << endl
             << "edge = " << edge << " count = " << edge_count << endl;
}

//--------------------------------------------------------------------------
// function: outport
// description: this function is an interface function between the simCode and the
// class to fire a projectile.  Calls the gunfire function.
//--------------------------------------------------------------------------

void SimbotClass::outport(int port_type, int port_num)
{
   if (port_type == 2) {
```

```
      if (port_num == 2) {
         gunfire();
      }
   }
}


//---------------------------------------------------------------------------
// function: range
// description: calculates the distance between the two Simbots.
//---------------------------------------------------------------------------

float SimbotClass::range(SimbotClass& botB)
{
   float range;
   range = ((botB.xpos-xpos)*(botB.xpos-xpos))
           + ((botB.zpos-zpos)*(botB.zpos-zpos));
   return (sqrt(range));
}


//---------------------------------------------------------------------------
// function: relative_angle
// description: translates the Simbot to the origin and then calculates the relative
// angle from itself to the other Simbot.
//---------------------------------------------------------------------------

float SimbotClass::relative_angle(SimbotClass& botB)
{
   float xbot = botB.xpos - xpos;
   float zbot = botB.zpos - zpos;

   float ang = atan2(xbot, zbot);
   ang = (rad2deg(ang));
   if (ang < 0) {
      ang += 360.0;
   }
   return ang;
}


//---------------------------------------------------------------------------
// These are two auxillary functions designed to convert between radians and
// degrees.   C++ uses radians for the trigonometric functions but the graphics
// package uses degrees.  Most of the comparisons of gun angles use degrees
// which the author feels is easier to visualize.
//---------------------------------------------------------------------------

float deg2rad (float degree)
{
   return (degree/180.0)*3.1415;
}


float rad2deg (float radian)
{
   return (radian/3.1415)*180.0;
}

//---------------------------------------------------------------------------
// end of SimbotClass.C file.
//---------------------------------------------------------------------------
```

APPENDIX D. SIMULATION CODE

The simulation main program is contained in the dual.C program that follows. The program has been provided without simulation code in place of the SimCode (also known as the process function) The SimCode is to placed in the process function, as discussed in Appendix E. The process function contains dummy code to allow the program to be compiled and even run, although the robots will not move. This code is not robust and does not allow the user to directly input any data except the end time of the simulation. To make changes to the Simbot objects the user must hard code the **set** and **setEyes** functions.

```
//-------------------------------------------------------------
// function: dual.C
// author: Doreen M. Jones
// date: May 31, 1997
// compiler: SUN/UNIX on SGI
// summary: this is the main driver function for the simulation requires
// two robot codes to properly run (not included here) This simulation
// sends the data from the Simbot movements to individual data files
// the simulation will dump the projectile information onto the screen
// but does not send this to the data files.  Uses UNIX system commands
// to facilitate the switching from one Simbot to the other.
//-------------------------------------------------------------

#include <sys/types.h>
#include <unistd.h>
#include <iostream.h>
#include <sys/prctl.h>
#include <sys/wait.h>
#include <signal.h>
#include <sys/param.h>
#include <iomanip.h>
#include <string.h>
#include <fstream.h>
#include <math.h>
#include "SimbotClass.h"

void processB(void *);
void platformA();
void position2();

SimbotClass botA;
SimbotClass c3p0;
int flag = 1;

int PIODA = 1;
int PIODB = 2;

int producer_pid;
int consumerA_pid;
int consumerB_pid;

int master_timerA = 0;
int master_timerB = 0;
int temp = 0;
ofstream outBot("BotAscript.dat", ios::out);
ofstream outBot2("BotBscript.dat", ios::out);

int main()
{
   int limit;
   cout << "enter end limit" << endl;
   cin >> limit;

   // botA is the grey simbot
   botA.set(-600.0, -600.0, 15, 4, 90.0, 800.0, 0);
   c3p0.set(600.0, -600.0, 15, 9, 90.0, 800.0, -1);
   strcpy (botA.buf, "0fgr0fgr");
   strcpy (c3p0.buf, "ffgrffgr");

   botA.setEyes(1.0, 1.0, 1.5, 2040.0, 3.0, 16.0);
   c3p0.setEyes(1.0, 1.0, 1.5, 2040.0, 3.0, 16.0);

   void processA(void *);
```

```
    producer_pid = getpid();
    consumerA_pid = sproc(processA, PR_SALL);
    consumerB_pid = sproc(processB, PR_SALL);

    setblockproccnt(consumerA_pid,0);
    setblockproccnt(consumerB_pid,0);
    setblockproccnt(producer_pid,0);

    flag = 1;
    unblockproc(consumerA_pid);

    while(temp < limit){
       while (flag) {
          blockproc(producer_pid);
       }

       flag = 1;
       unblockproc(consumerB_pid);

       while (flag) {
          blockproc(producer_pid);
       }
       flag = 1;

       unblockproc(consumerA_pid);
       temp++;
    }

    cout<< "end game "<< endl;
    cout << "botA stats = " << endl;
    for (int ii = 14; ii >= 0; ii--) {
       for (int jj = 0; jj < 8; jj++) {
          cout << botA.bullets[ii][jj] << ' ' ;
       }
       cout << endl;
    }
    cout << "botB stats = " << endl;
    for ( ii = 14; ii >= 0; ii--) {
       for (int jj = 0; jj < 8; jj++) {
           cout << c3p0.bullets[ii][jj] << ' ' ;
       }
       cout << endl;
    }
    kill(consumerB_pid, SIGKILL);
    kill(consumerA_pid, SIGKILL);
    return 0;

}


//-------------------------------------------------------------------------
// function: processA
// description: this is where the simCode for Simbot A will be placed.
//  this function holds the simCode and steps through it in a similar manner
//  as the robot does.
//-------------------------------------------------------------------------


void processA(void *)
{
   platformA();
   return;
}
```

```
// ------------------------------------------------------------------------
// description: platformA
// description: acts like the Dynamic C platform function in that it will send
// the information to the motor controller but it also sends the data to the
// output file and sets the flags to hand off to control to the other Simbot.
// ------------------------------------------------------------------------

void platformA()
{
   while (!flag) {
      blockproc(consumerA_pid);
   }

   ofstream outBot("BotAscript.dat", ios::app);
   outBot << setiosflags(ios::fixed) << setprecision(2) ;
   outBot << setw(6) << master_timerA++ << ' '
          << setw(6) << (botA.theta-90) << ' '
          << setw(2) << botA.color << ' '
        << setw(7) << botA.xpos << ' ' << botA.zpos << ' '
          << endl;


   botA.platform(c3p0);

   flag = 0;
    unblockproc(producer_pid);
}

//------------------------------------------------------------------------
// function: processB
// description: this is where the simCode for Simbot B will be placed.
//  this function holds the simCode and steps through it in a similar manner
//  as the robot does.
//------------------------------------------------------------------------

void processB(void *)
{
   position2();
   return;
}


// ------------------------------------------------------------------------
// description: position2
// description: acts like the Dynamic C platform function in that it will send
// the information to the motor controller but it also sends the data to the
// output file and sets the flags to hand off to control to the other Simbot.
// ------------------------------------------------------------------------

void position2()
{

   while (!flag) {
      blockproc(consumerB_pid);
   }
   ofstream outBotB("BotBscript.dat", ios::app);

   outBot2 << setiosflags(ios::fixed) << setprecision(2);
   outBot2 << setw(6) << master_timerB++ << ' '
           << setw(6) << (c3p0.theta-90) << ' '
           << setw(2) << c3p0.color << ' '
         << setw(7) << c3p0.xpos << ' ' << c3p0.zpos << ' '
           << endl;
```

```
    c3p0.platform(botA);

    flag = 0;
    unblockproc(consumerA_pid);
    unblockproc(producer_pid);
}

//--------------------------------------------------------
// end of file dual.C
//--------------------------------------------------------
```

APPENDIX E. DYNAMIC C CONVERSION

The following is required for the initial setup before the conversion can be accomplished.

i) At least one working Dynamic C program (i.e., compiles and runs the robot), two working programs are better.

ii) The Dynamic C program must be written with while loops.

iii) Identify which robot is botA and which is botB.

iv) Determine how many PIODA and PIODB CALLS are made and what they do. The simulation in its present state only allows for a fire signal identified as 0x02.

v) Check the for loops, they need to be incrementing and use the same counter for the ones that control the amount of time the robot will move for.

vi) The simulation only allows for one set of eyes. In the **ad_rd8(int n)** function, n=0 is the left eye and n=1 is the right eye.

vii) Identify the **#define** statements used and determine their importance. In the present simulation, **#define** statements can work, but you have to make sure they do not conflict with any others in the main program and in other robot code.

viii) Create function prototypes for any additional functions used in the code (besides the **platform** and **interrupt** functions). The function prototypes need to be inserted before the related **process** function.

ix) All comment lines need to start with // instead of the C style block commands /* or just remove all comments.

The following steps will guide you through the robot code conversion process.

1. The first program to be converted is **botA**.

2. Using the text editor search and replace command, change the following function calls to reflect the Simbot class requirements.

    a. Change **platform** to **platformA**.
    b. Change **outport** to **botA.outport**.
    c. Change the counter variable name to **botA.count**.
    d. Change **buf** to **bot.buf**.
    e. Change **ad_rd8(0)** to **botA.ad_rd8(0, botB)**.

f. Change **ad_rd8(1)** to **botA.ad_rd8(1, botB)**.

3. Place any **#define** statements used at the top of the program. (Note: if the program does not compile because of this, consider eliminating them altogether.)

4. Remove any hardware setup code lines.

5. Remove any **#use** or **#include** lines.

6. Remove the **platform** and **interrupt** functions.

7. Create the function prototypes and place them before **main**.

8. Remove any extraneous code lines above **main** (i.e., hardware setup lines).

9. Exchange the for loop and **platformA** lines and remove the semicolon after the for loop line (see Figure 5.1).

10. Rename **main( )** as **processA(void *)**.

11. Save to a separate file.

12. Find the **processA/SimCodeA** place holder in **dual.C**. Highlight the place holder and copy/insert the file created in step 11 over the place holder.

13. Repeat the steps above for the other robot, substituting **B** for **A**.

Not all situation have been incorporated into the current simulation. The robot code may have functionality that has not been incorporated into the simulation. For instance, gun motors are considered to be on at all times. The converted code that follows was created from the Dynamic C code found in Appendix B.

APPENDIX F. USING THE SIMULATION

Once the robot code has been converted to Simcode and assimilated into dual.C, the programs must be compiled. At the command prompt, type **make dual**. If the program fails to compile it may mean one of several things. The most prevalent errors will be from failure to remove all C style comments. After debugging, and remaking, type **dual** at the command prompt. Run the simulation for the desired length of time, 2000-5000 provide several minutes of simulation. Values less than 500 are not very beneficial. **Dual** is complete, type **show_wars** at the command prompt. Enter the desired start and stop times after the prompts. The simulation graphics windows will open on the screen. Additional functionality is found in the second window. The graphics portions is designed with 6 different view angles and replay ability. The program will create two output data files: **botAscript.dat** and **botBscript.dat**. The files contain the time count, rotation angle, x-position and z-position.

To run the graphical program, type **show_war** at the prompt. The program will be expecting to find the two files listed above and will not run if the programs are not available to it in the same directory. Then type in the start and stop times to view. This code is not robust, but it will not proceed if you did not type in a stop time that is higher than the start time. Once the animation has started you can use the camera buttons to change the viewing angle. The **pause** button will stop the animation at that frame until you hit the **move** button. The **replay** button will reshow the portion of the animation previously shown and the **restart** button will request new start and stop times in the command window. The **floor** options changes the type of flooring to slow down the display. Currently there are only two type of floors, the fast and the slow. The neutral option will display the fast floor. The **end** button will exit the program.

APPENDIX G. SIMULATION ANIMATION CODE

The simulation animation or graphics package is contained in ten files. The animation program uses the data files, **Bot1script.dat** and **Bot2script.dat** created by the dual.C program. A brief description of the graphics files is include below.

1. **show_war_funcs.h** - header file for show_wars.C containing the function prototypes

2. **show_war.C** - main animation file that controls the animation of the Simbot Wars

3. **materialsupport_funcs.h** - contains the function prototypes for the material colors used in the

4. **materialsupport.h** - contains the variables assigning color names to integer for use in the draw support functions

5. **materialsupport.C** - contains the functions to create the colors used in the animation

6. **drawsupport_funcs.h** - contains the function prototypes for drawsupport.C and the prototypes for drawBot.C.

7. **drawsupport.C** - contains the OpenGL primitives to create the playing field and the Simbots in the animation

8. **drawBot.C** - contains the 3-D Simbot model created from OpenGL primitives for use in show_wars.C.

```
//-------------------------------------------------------------
//FILENAME: show_war.h
//AURTHOR: Doreen M. Jones (derived from M. Zyda class notes.
//DATE: 31 MAY 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY: This is the header file for show_war.C
//-------------------------------------------------------------

static void initCB (Widget w, XtPointer, XtPointer call_data);
static void exposeCB(Widget w, XtPointer, XtPointer );
static void resizeCB (Widget w, XtPointer, XtPointer call_data );
static void inputCB(Widget, XtPointer, XtPointer call_data );

static void quitCB(Widget w, XtPointer, XtPointer);
static void resetCB(Widget, XtPointer, XtPointer);

static void ReplayCB(Widget w, XtPointer, XtPointer);
static void Types_of_lightsCB(Widget w, XtPointer, XtPointer);
static void movementCB(Widget w, XtPointer, XtPointer);

static void camera1CB(Widget, XtPointer, XtPointer);
static void camera2CB(Widget, XtPointer, XtPointer);
static void camera3CB(Widget, XtPointer, XtPointer);
static void camera4CB(Widget, XtPointer, XtPointer);
static void camera5CB(Widget, XtPointer, XtPointer);
static void camera6CB(Widget, XtPointer, XtPointer);

GLboolean drawWP();
void draw_the_scene();
void make_the_window_for_the_controls();
void create_toggle_widget(Widget parent);

void create_nine_entry_selector (
   Widget rc,              // Parent widget is a RowColumn container.
   char *radioboxname,     // Text name to use for the RadioBox.
   char *all_labels[],     // All the labels for the selector.
   XtCallbackProc all_callbacks[],  // All the callbacks for each
                                    // option.
   XtPointer all_data[],   // Pointers to the data to be passed
                           // to the callbacks.
   Widget all_toggles[]); // Returned toggle widgets.


void initialize_toggle_states();
void build_accumulative_matrices();
void reset_show();

static void slow_floorCB(Widget, XtPointer user_data, XtPointer);
static void neutralCB(Widget, XtPointer user_data, XtPointer);
static void fast_floorCB(Widget, XtPointer user_data, XtPointer);

//-------------------------------------------------------------
// end of file show_war.h
//-------------------------------------------------------------
```

```
//-------------------------------------------------------------------
//FILENAME: show_war.C
//AUTHOR: Doreen M Jones (derived from M. Zyda class notes
//DATE: 31 MAY 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY: This is the main driver program for the simulation
// graphics display.  It requires the direct use of the drawbot.C
// and drawsupport.C files to create the objects in the scene.
// this was designed to maximize the animation speed vice the
// a more detailed scene.  As this is a derived program, some
// latent functionality remains that is not used by the program.
//-------------------------------------------------------------------

#include <Xm/Xm.h>           // Get the Motif stuff...
#include <Xm/Form.h>         // We are going to use a Form container widget.
#include <Xm/Frame.h>        // Get the Frame widget.
#include <Xm/RowColumn.h>    // Get the RowColumn widget.
#include <Xm/CascadeB.h>     // Get the CascadeButton widget.
#include <Xm/PushB.h>        // Get the PushButton widget.
#include <Xm/ToggleB.h>      // Get the ToggleButton widget.
#include <Xm/Separator.h>     // Get the Separator widget.
#include <GL/GLwMDrawA.h>    // We are going to use an OpenGL Motif Draw widget
#include <X11/StringDefs.h>
#include <X11/keysym.h>
#include <GL/gl.h>           // Get the OpenGL required includes.
#include <GL/glu.h>
#include <GL/glx.h>
#include <iostream.h>               // C++ I/O subsystem.
#include <math.h>
#include <stdlib.h>                 // Get exit() function.
#include <fstream.h>

#include "show_war_funcs.h"          // Get this program's function declarations.
#include "materialsupport.h"        // Get material names.
#include "materialsupport_funcs.h"  // Get the material support functions.
#include "drawsupport_funcs.h"      // Get the drawing functions.

static XtAppContext app_context;        // Applications context.
static XtWorkProcId workprocid = NULL;  // Id of the work procedure
static GLXContext glx_context;          // A GL context (see initCB).
Display *global_display;                // Global display.
Window global_window;                   // Global window.
XmStringCharSet charset = (XmStringCharSet) XmSTRING_DEFAULT_CHARSET;

static float dummy; // dummy variable for widgets
extern float FloorSize = 1500.00; // default value
void All_lights_on();

// camera values
static float frog1 = 2000;
static float frog2 = FloorSize;
static float frog3 = -511;
static float toad1 = 0.0;
static float toad2 = 0.0;
static float toad3 = -511.0;

GLboolean lightType = FALSE;
GLboolean slow_floor = FALSE;
GLboolean fast_floor = FALSE;
GLboolean StartUp = FALSE;

static int eyepoint = 0;
static int floorStyle = 1 ;
static int start_time = 0;
```

```
static int end_time;
static int reset = 1;
static int pause = 0;
    Arg wargs[15];   // Args used with XtSetArg below.
    int n;           // Used to specify number of args.


//--------------------------------------------------------------
//function: main
//description: starts up the GUI and begins drawing the scene
// which continues until told to stop.
//--------------------------------------------------------------

void main(int argc, char **argv)
{
    Arg wargs[15];   // Args used with XtSetArg below.
    int n;           // Used to specify number of args.
    Widget toplevel; // The top level (shell) widget.
    Widget form;     // The topmost container widget (excluding the shell).
                     // This type of widget is used as a container for a
                     // collection of widgets that dynamically resize
                     // together.
    Widget frame;    // Frame widget attached to Form widget.
    Widget glw;      // The GL widget that we use to draw into.
    // Fallback resources for the application.
    static char * fallback_resources[] = {
        "*form*background:             SGISlateBlue",  // /usr/lib/rgb.txt
        "*frame*shadowType:            SHADOW_IN",
        "*glwidget*width:              750",
        "*glwidget*height:             600",
        "*glwidget*rgba:               TRUE",
        "*glwidget*doublebuffer:       TRUE",
        "*glwidget*allocateBackground: TRUE",
        "*fontList: -adobe-courier-bold-r-normal--12-120-75-75-m-70-iso8859-1",
        "Mylight*geometry: 750x600+265+370", // width, height, xoff, yoff
        "Shell2*geometry: 250x475+1020+0",
        "Shell2*background:            SGISlateBlue",
        NULL
    };
    // Create the top level widget.
    toplevel = XtAppInitialize(
       &app_context,                  // Application context.
       "RobotWars",                     // Application class.
       NULL, 0,                       // Command line option list.
       &argc, argv,                   // Command line args.
       fallback_resources,            //
       NULL ,                         // Argument list.
       0);                            // Number of arguments.
   // Create Form widget.
   // This is the top level container for below widgets.
   n = 0;
   form = XmCreateForm(toplevel, "form", wargs, n);
   // Create a Frame widget to make it so we can resize the window.
   n = 0;
   XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
   XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
   XtSetArg(wargs[n], XmNrightAttachment, XmATTACH_FORM); n++;
   XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
   frame = XmCreateFrame (form, "frame", wargs, n);
   // Create a GL widget.
   n = 0;
   XtSetArg(wargs[n], GLwNdepthSize, 1); n++;
   glw = GLwCreateMDrawingArea(frame, "glwidget", wargs, n);
   // Add callbacks to the glw widget.
```

```
    XtAddCallback(glw, GLwNginitCallback,  initCB,   (XtPointer) NULL);
    XtAddCallback(glw, GLwNexposeCallback, exposeCB, (XtPointer) NULL);
    XtAddCallback(glw, GLwNresizeCallback, resizeCB, (XtPointer) NULL);
    // Add in a callback for processing input keys from the keyboard.
    XtAddCallback(glw, GLwNinputCallback, inputCB, (XtPointer) NULL);

    // Add in the work procedure.
    // A work procedure is called repeatedly whenever there are no
    // events to process.

    workprocid = XtAppAddWorkProc(app_context, (XtWorkProc)drawWP,
                                  (XtPointer)NULL);

    // Create the control panel for the application...
    make_the_window_for_the_controls();
    // Manage all the child widgets.
    // We delayed managing and did it in reverse order to get the right
    // size widgets automatically.
    XtManageChild(glw);
    XtManageChild (frame);
    XtManageChild(form);
    XtRealizeWidget(toplevel);  // Instantiate it now.
    XtAppMainLoop(app_context); // Loop for events.

}   // end of main().


//--------------------------------------------------------------------------
//function: initCB
//description: This function initializes graphics modes and transformation matrices.
//--------------------------------------------------------------------------

static void initCB (Widget w, XtPointer, XtPointer call_data)
{
    // Get us a GLwDrawingAreaCallbackStruct ptr to the call_data.
    GLwDrawingAreaCallbackStruct *glptr = (GLwDrawingAreaCallbackStruct *) call_data;
    Arg wargs[1];       // Arg temp.
    XVisualInfo *vi;    // Pointer to XVisualInfo.
    GLint gdtmp;        // Used in the get capability stuff below.

    // Get the visual info...
    XtSetArg(wargs[0], GLwNvisualInfo, &vi);
    XtGetValues(w, wargs, 1);

    // Create a new GLX rendering context.
    // GL_TRUE -> Specify direct connection to graphics system (if possible).
    glx_context = glXCreateContext(XtDisplay(w), vi, 0, GL_TRUE);

    // Set the global window and display.
    global_display = XtDisplay(w);
    global_window  = XtWindow(w);

    // Make this drawing area the current one.
    GLwDrawingAreaMakeCurrent(w, glx_context);

    // Test to see if this machine has a z-buffer.
    if((glGetIntegerv(GL_DEPTH_BITS, &gdtmp), gdtmp) == 0)
    {
        cerr <<  "This machine does not have a hardware zbuffer" << endl;
        exit(0);
    }

    // Turn on z-buffering.
    glEnable(GL_DEPTH_TEST);
```

```
   // Turn on Gouraud shading.
   glShadeModel(GL_SMOOTH);

   // Set the Viewport for the first draw of the window.
   glViewport (0, 0, glptr->width, glptr->height);

   reset_show();
   All_lights_on();

}  // end of initCB()

//---------------------------------------------------------------------
// function: exposeCB -
// description:  This routine is called whenever the window is uncovered.
//---------------------------------------------------------------------

static void exposeCB(Widget w, XtPointer, XtPointer )
{
   // Set the window into which GL drawing should be done.
   GLwDrawingAreaMakeCurrent(w, glx_context);

   // Draw the scene.
   draw_the_scene();
}


//---------------------------------------------------------------------
// function: resizeCB -
// description: This routine is called whenever the window is moved or resized.
//---------------------------------------------------------------------

static void resizeCB (Widget w, XtPointer, XtPointer call_data )
{
   // Get us a GlwDrawingAreaCallbackStruct ptr to the call_data.
   GLwDrawingAreaCallbackStruct *glptr = (GLwDrawingAreaCallbackStruct *) call_data;

   // Set the window into which GL drawing should be done.
   GLwDrawingAreaMakeCurrent(w, glx_context);

   // Set the viewport using the window size currently set.
   glViewport (0, 0, (GLsizei) glptr->width, (GLsizei) glptr->height);

   // Draw the scene.
   draw_the_scene();
}



//---------------------------------------------------------------------
// funciton: inputCB -
// description:  This routine handles all types of input from the GL widget.
//   In this particular example, the KeyRelease handles the
//   Escape-Key so that its release exits the program.
//---------------------------------------------------------------------

static void inputCB (Widget, XtPointer, XtPointer call_data )
{
   // Get us a GLwDrawingAreaCallbackStruct ptr to the call_data.
   GLwDrawingAreaCallbackStruct *glptr = (GLwDrawingAreaCallbackStruct *) call_data;

   char ascii[1];    // Hold for the ascii chars returned from XLookupString.

   int nchars;       // Number of characters returned from XLookupString.
```

```
   KeySym keysym;    // The X version of the returned character.

   XKeyEvent *ptr;  // ptr to the Key Event structure.

   // We look at the type of event to see if we should pay attention...
   // In this example, we only pay attention to key release events.
   switch(glptr->event->type)
   {
      case KeyRelease:
         // We must convert the keycode to a KeySym before it is possible
         // to check if it is an escape. We also dump the Ascii into the
         // array ascii. The return value from XLookupString is the
         // number of characters dumped into array ascii.
         ptr = (XKeyEvent *) glptr->event;
         nchars = XLookupString(ptr, ascii, 1, &keysym, NULL);

         if(nchars == 1 && keysym == (KeySym) XK_Escape)
         {
            // We have an escape. Time to exit.
            exit(0);
         }

         break;

      default:
         break;
   }   // end of switch on event->type.

}   // end of inputCB.



//---------------------------------------------------------------------------
// function: make_the_window_for_the_controls
// description: This function creates a second window for the control panel.
//---------------------------------------------------------------------------

void make_the_window_for_the_controls()
{

   Arg wargs[16];    // Args used with XtSetArg below.
   int n;            // Used to specify number of args.
   Widget shell2;    // Second shell widget for the controls.
   Widget controls; // Form widget for the toggle controls.
   Display *dpy;     // The X Display...
   static int zero = 0;

   // Open a connection to the X server.
   dpy = XtOpenDisplay(app_context, NULL, NULL, "Shell2", NULL, 0,
                       &zero, NULL);

   // Create a second applications shell.
   n = 0;
   shell2 = XtAppCreateShell(NULL, "Shell2",
                             applicationShellWidgetClass,
                              dpy, wargs, n);

   // Create a Form widget for the controls...
   n = 0;
   XtSetArg(wargs[n], XmNwidth, 480); n++;
   XtSetArg(wargs[n], XmNheight, 290); n++;
   controls = XmCreateForm(shell2, "controls", wargs, n);

   // Create the Toggle widgets.
```

```
    create_toggle_widget(controls);

    // Manage after creation ...
    XtManageChild(controls);

    // Realize the second shell widget.
    XtRealizeWidget(shell2);

}

// Data for the option selector.
static char *all_option_labels[] = { "Camera 1", "Camera 2", "Camera 3",
                                     "Camera 4", "Camera 5", "Camera 6",
                                     "slow_floor", "neutral", "fast_floor",
                                   };


static void ( *all_option_callbacks[] )(Widget, XtPointer, XtPointer) =
                                  {   camera1CB, camera2CB, camera3CB,
                                      camera4CB, camera5CB, camera6CB,
                                      slow_floorCB, neutralCB, fast_floorCB,
                                  };

static Widget all_option_toggles[16];   // All returned option toggle widgets.

static XtPointer all_translate_data[] = { (XtPointer) &dummy,
                                          (XtPointer) &dummy,
                                          (XtPointer) &dummy,
                                           (XtPointer) &dummy,
                                           (XtPointer) &dummy,
                                          (XtPointer) &dummy  };

//--------------------------------------------------------------------------
//function:  create_toggle_widget (Widget parent)
//description: Input parent is expected to be a Form widget.
//    This function creates 4 different RowColumn widgets,
//    each RowColumn to hold the toggle buttons corresponding to
//    rotation, translation & scaling and reset & exit.
//--------------------------------------------------------------------------

void create_toggle_widget (Widget parent)
{

   Widget rc[4];     // We need 4 RowColumn widgets.
   Widget button;    // PushButton widget.
   XmString label_string;   // String.
   Arg wargs[10];  // Same old args stuff.
   int n;          // Same old args stuff.

   // Create a RowColumn widget container
   n = 0;
   rc[0] = XmCreateRowColumn(parent, "rowcol", wargs, n);
   XtManageChild(rc[0]);

   // Define a nine_entry selector for rotation.
   create_nine_entry_selector(rc[0], "option",
                              all_option_labels,
                              all_option_callbacks,
                              all_translate_data,
                              all_option_toggles);

   // Create a RowColumn widget container
   n = 0;
   XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
```

```
XtSetArg(wargs[n], XmNleftWidget, rc[0]); n++;
rc[1] = XmCreateRowColumn(parent, "rowcol", wargs, n);
XtManageChild(rc[1]);

// Create a RowColumn widget container
n = 0;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(wargs[n], XmNleftWidget, rc[1]); n++;
rc[2] = XmCreateRowColumn(parent, "rowcol", wargs, n);
XtManageChild(rc[2]);

// Create a RowColumn widget container
n = 0;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(wargs[n], XmNleftWidget, rc[2]); n++;
rc[3] = XmCreateRowColumn(parent, "rowcol", wargs, n);
XtManageChild(rc[3]);


// Now add a Reset PushButton.
n = 0;
//  XtSetArg(wargs[n], XmNalignment, XmALIGNMENT_CENTER); n++;
label_string = XmStringCreateLtoR("Reset", charset);
XtSetArg(wargs[n], XmNlabelString, label_string); n++;
button = XmCreatePushButton(rc[3], "reset", wargs, n);
XtManageChild(button);
XmStringFree(label_string);

// Here is a reset callback...
XtAddCallback (button, XmNarmCallback, resetCB, (XtPointer) NULL);

// Now add an Exit PushButton.
n = 0;
//  XtSetArg(wargs[n], XmNalignment, XmALIGNMENT_CENTER); n++;
label_string = XmStringCreateLtoR("Exit", charset);
XtSetArg(wargs[n], XmNlabelString, label_string); n++;
button = XmCreatePushButton(rc[3], "exit", wargs, n);
XtManageChild(button);
XmStringFree(label_string);

// Now add an Exit callback for the PushButton.
XtAddCallback(button, XmNarmCallback, quitCB, (XtPointer) NULL);

n = 0;
// XtSetArg(wargs[n], XmNalignment, XmALIGNMENT_CENTER); n++;
label_string = XmStringCreateLtoR("Replay", charset);
XtSetArg(wargs[n], XmNlabelString, label_string); n++;
button = XmCreatePushButton(rc[3], "Replay", wargs, n);
XtManageChild(button);
XmStringFree(label_string);

// Now add a callback for the PushButton.
XtAddCallback(button, XmNarmCallback, ReplayCB, (XtPointer) NULL);

// Now add an movement PushButton.
n = 0;
// XtSetArg(wargs[n], XmNalignment, XmALIGNMENT_CENTER); n++;
label_string = XmStringCreateLtoR("Pause", charset);
XtSetArg(wargs[n], XmNlabelString, label_string); n++;
button = XmCreatePushButton(rc[2], "Pause", wargs, n);
XtManageChild(button);
XmStringFree(label_string);

// Now add an movement callback for the PushButton.
```

```
   XtAddCallback(button, XmNarmCallback, movementCB, (XtPointer) NULL);

}   // end of create_toggle_widget...


//-------------------------------------------------------------------------
//function :  create_nine_entry_selector
//description:   This function creates a nine entry selection box for the
//  camera and the floor types.
//-------------------------------------------------------------------------

void create_nine_entry_selector (
   Widget rc,              // Parent widget is a RowColumn container.
   char *radioboxname,     // Text name to use for the RadioBox.
   char *all_labels[],     // All the labels for the selector.
   XtCallbackProc all_callbacks[],  // All the callbacks for each
                                    // option.
   XtPointer all_data[],    // Pointers to the data to be passed
                            // to the callbacks.

   Widget togglereturns[16]) // Returned toggle button widgets.

{
   Widget myradio[4];        // RadioBox widgets.
   Widget separator;         // Separator widget for the RadioBox.
   Arg wargs[10];            // Same old args stuff.
   int n;                    // Same old args stuff
   int i=0,j=0;                  // Loop temps.

   for(j=0; j < 4; j++) {
      n = 0;
      XtSetArg (wargs[n], XmNentryClass, xmToggleButtonWidgetClass); n++;
      myradio[j] = XmCreateRadioBox (rc, radioboxname, wargs, n);
      XtManageChild (myradio[j]);
      if (j < 1 ) {
         for (i = 0 ; i < 6; i++) {
            togglereturns[i] = XtCreateManagedWidget(all_labels[i],
                                                     xmToggleButtonWidgetClass,
                                                     myradio[j],
                                                     NULL, 0);
            // Add a callback for this toggle button.
            XtAddCallback (togglereturns[i], XmNarmCallback, all_callbacks[i],
                           all_data[i]);
         }   // end for i ...
         n=0;
         separator = XmCreateSeparator(rc, "separator", wargs, n);
         XtManageChild(separator);
      }
      else if (j == 1 ) {
         for (i = 6 ; i < 9; i++) {
            togglereturns[i] = XtCreateManagedWidget(all_labels[i],
                                                     xmToggleButtonWidgetClass,
                                                     myradio[j],
                                                     NULL, 0);
            // Add a callback for this toggle button.
            XtAddCallback (togglereturns[i], XmNarmCallback, all_callbacks[i],
                           all_data[i]);
         }   // end for i ...
         n=0;
         separator = XmCreateSeparator(rc, "separator", wargs, n);
         XtManageChild(separator);
      }
   }   // end for j
```

```
}

//--------------------------------------------------------------------------
// function: initialize_toggle_states
// description: This function sets the toggles to their initial depressions.
//--------------------------------------------------------------------------

void initialize_toggle_states()
{
   Arg wargs_false[16];  // Same old args stuff.
   Arg wargs_true[16];   // Same old args stuff.
   int n;          // Same old args stuff.

   // Make the argument list that says False and one that says true.
   n = 1;
   XtSetArg(wargs_false[0], XmNset, False);
   XtSetArg(wargs_true[0], XmNset, True);

   // Set all the camera toggles.
   XtSetValues(all_option_toggles[0], wargs_true, n);
   XtSetValues(all_option_toggles[1], wargs_false, n);
   XtSetValues(all_option_toggles[2], wargs_false, n);

   XtSetValues(all_option_toggles[3], wargs_false, n);
   XtSetValues(all_option_toggles[4], wargs_false, n);
   XtSetValues(all_option_toggles[5], wargs_false, n);
   //set the floor toggles
   XtSetValues(all_option_toggles[6], wargs_false, n);
   XtSetValues(all_option_toggles[7], wargs_true, n);
   XtSetValues(all_option_toggles[8], wargs_false, n);
}


//--------------------------------------------------------------------------
// function: quitCB -
// description:  This function is called whenever the "Quit" menu option
//   is selected.
//--------------------------------------------------------------------------

static void quitCB (Widget w, XtPointer, XtPointer)
{
    // Bye, bye...
    XtCloseDisplay(XtDisplay(w));
    exit(0);
}   // End of quitCB



//--------------------------------------------------------------------------
// function: resetCB -
// description: This function is called whenever the "Reset" menu option
//   is selected.
//--------------------------------------------------------------------------

static void resetCB (Widget, XtPointer, XtPointer)
{
   reset = 1;
   reset_show();
}   // End of resetCB
```

```
//------------------------------------------------------------------------
// function: ReplayCB -
// description: This function is called whenever the "Replay" menu option
//   is selected.
//------------------------------------------------------------------------

static void ReplayCB (Widget, XtPointer, XtPointer)
{
   reset = 2;
   reset_show();
}


//------------------------------------------------------------------------
// function: movementCB
// description: This function is called whenever the "pause" menu option
//   is selected.
//------------------------------------------------------------------------

static void movementCB (Widget w, XtPointer, XtPointer)
{
   if (pause == 1) {
      pause = 0;
      n = 0;
      XtSetArg(wargs[n], XmNlabelString,
      XmStringCreateLtoR("pause", XmFONTLIST_DEFAULT_TAG)); n++;
      XtSetValues(w, wargs, n);
   }
   else {
      pause = 1;
      n = 0;
      XtSetArg(wargs[n], XmNlabelString,
      XmStringCreateLtoR(" move ", XmFONTLIST_DEFAULT_TAG)); n++;
      XtSetValues(w, wargs, n);
   }
}



//------------------------------------------------------------------------
//function: drawWP -
//description:   This function is called by the work procedure.
//   It is called repeatedly whenever there are no events to process.
//   This function returns FALSE so that the work procedure does NOT
//   stop calling it.
//------------------------------------------------------------------------

GLboolean drawWP()
{
    // Draw the scene again...
    draw_the_scene();
    // If we do not return FALSE, the work procedure will stop calling
    // this function.
    return(FALSE);
}
```

```
//-----------------------------------------------------------------------------
//function: draw_the_scene
//   This function draws the picture.
//-----------------------------------------------------------------------------
//
void draw_the_scene()
{
   int type = 0;
   char* name = "BotAscript.dat";
   char* name2 = "BotBscript.dat";
   glClearColor(0.1, 0.1, 0.1, 0.0);
   glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0,  1.25,  1.0,  10000.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   if (eyepoint == 0) {
      gluLookAt(frog1, frog2, frog3,    // View point. changed 100 to 500
                toad1, toad2, toad3,   // Ref point, point we are looking towards.
                0.0, 1.0, 0.0 );
   }

   turnOnTheLightModel();
   type = lightType;
   turnOnTheLights(type);

   if (slow_floor) {
      drawthefloor2(0.0, -200.0, -511.0, 1600.0);}
   else
      drawthefloor3(0.0, -200.0, -511.0, 1600.0);

   glMatrixMode(GL_MODELVIEW);

   PaintBotB(start_time, end_time, name2, pause, reset);
   PaintBotA(start_time, end_time, name, pause, reset);

   turnOffTheLights();
   turnOffTheLightModel();

   // Swap the buffers as we are doing double buffering.
   glXSwapBuffers(global_display, global_window);

}   // end of draw_the_scene


//-----------------------------------------------------------------------------
// function: reset_show
// The following routine sets all accumulative matrices to unit matrices.
// This routine also resets the toggle button states.
//-----------------------------------------------------------------------------

void reset_show()
{
   slow_floor = FALSE;
   fast_floor = FALSE;
   floorStyle = 0;

   if (reset == 1) {
     end_time = 0;
     cout << "Enter start time" << endl;
     cin >> start_time ;
     if (start_time < 1) {
```

```
         start_time = 1;
      }
      while (end_time <= start_time) {
         cout << "Enter end time" << endl;
         cin >> end_time;
         end_time --;
      }
      reset = 0;
   }
   // Set all the toggle buttons to their initial states.
   initialize_toggle_states();
}


//--------------------------------------------------------------------------
// These next three call backs will set the floor type used in the
// scene.  Only two floor types are available but three have been set
// up for additional functionality that can be added at a later date.
//--------------------------------------------------------------------------
//function: slow_floorCB
//description: draws the 64 square checker floor
//--------------------------------------------------------------------------

static void slow_floorCB(Widget, XtPointer , XtPointer)
{
   slow_floor = TRUE; fast_floor = FALSE;
}

//--------------------------------------------------------------------------
//function: fast_floorCB
//description: draws a single square floor
//--------------------------------------------------------------------------

static void fast_floorCB(Widget, XtPointer, XtPointer)
{
   fast_floor = TRUE;  slow_floor = FALSE;
}

//--------------------------------------------------------------------------
//function: neutralCB
//description: draws a single square floor, this was left for future use
// for the 16 square floor or to add in the sensor rings.
//--------------------------------------------------------------------------

static void neutralCB(Widget, XtPointer, XtPointer)
{
   fast_floor = TRUE;  slow_floor = FALSE;
   // this will be another floor style
}

//--------------------------------------------------------------------------
// These next six call backs will set the camera angle.  These angles have
// not been set for any abitrary reason.  The eyepoint was used to draw a
// camera on the robot but has been removed at this time.  Camera1 is the
// default camera and cameras 5 and 6 are at the level of the playing
// surface.
//--------------------------------------------------------------------------

static void camera1CB(Widget, XtPointer, XtPointer)
{
   frog3 = -511; frog2 = FloorSize; frog1 = 2000;
   toad3 = -511; toad2 = 0.0; toad1 = 00.0;
   eyepoint = 0;
}
```

```
static void camera2CB(Widget, XtPointer, XtPointer)
{
   frog1 = 0.0; frog2 = FloorSize; frog3 = 2000-511;
   toad1 = 0.0; toad2 = 0.0; toad3 = -511;
   eyepoint = 0;
}

static void camera3CB(Widget, XtPointer, XtPointer)
{
   frog1 = -2000.0; frog2 = FloorSize; frog3 = -511;
   toad1 = 0.0; toad2 = 0.0; toad3 = -511.0;
   eyepoint = 0;
}

static void camera4CB(Widget, XtPointer, XtPointer)
{
   frog1 = 0.0; frog2 = 1500.0; frog3 = -2500-511; //739
   toad1 = 0.0; toad2 = 0.0; toad3 = 0;  //-1775
   eyepoint = 0;
}

static void camera5CB(Widget, XtPointer, XtPointer)
{
   frog1 = -FloorSize+250; frog2 = 0.0; frog3 = -511.0;
   toad1 = FloorSize-250; toad2 = 0.0; toad3 = -511.0;
   eyepoint = 0;
}

static void camera6CB(Widget, XtPointer, XtPointer)
{
   eyepoint = 1;
}


//-------------------------------------------------------------------------
//function:A11_lights_on
//description: turns on the lights in the scene.
//-------------------------------------------------------------------------

void A11_lights_on()
{
   glEnable(GL_LIGHT0);
}

//-----------------------------------------------------------------
// end of file show_war.C
//-----------------------------------------------------------------
```

```
//-------------------------------------------------------------------
//FILENAME: materialsupport_funcs.h
//AUTHOR: Doreen M Jones (derived from M. Zyda class notes
//DATE: 31 MAY 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY: contains function prototypes for materialsupport.C
//-------------------------------------------------------------------

void turnOnTheLightModel();
void turnOffTheLightModel();
void turnOnTheLights(int type);
void turnOffTheLights();
void turnOnMaterial(GLenum whichFace, int whichMaterial);
void makeGLMaterialCalls(GLenum whichFace,
                         GLfloat *ambient,
                         GLfloat *diffuse,
                         GLfloat *specular,
                         GLfloat *shininess,
                         GLfloat *emmissive);


//-------------------------------------------------------------------
// end of file materialsupport_funcs.h
//-------------------------------------------------------------------
```

```
//-------------------------------------------------------------------
//FILENAME: materialsupport.h
//AUTHOR: Doreen M Jones (derived from M. Zyda class notes
//DATE: 31 MAY 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY: Defines a set of constant names for the materials.
//-------------------------------------------------------------------

#define BRASS  1
#define SHINYBRASS  2
#define PEWTER  3
#define SILVER  4
#define GOLD  5
#define SHINYGOLD  6
#define PLASTER  7
#define REDGLOW  8
#define GREENPLASTIC  9
#define BLUEPLASTIC  10
#define RED  11
#define WHITE  12
#define BLACK 13

// Define a set of locations we can use...

static int _BRASS = BRASS;
static int _SHINYBRASS = SHINYBRASS;
static int _PEWTER = PEWTER;
static int _SILVER = SILVER;
static int _GOLD = GOLD;
static int _SHINYGOLD = SHINYGOLD;
static int _PLASTER = PLASTER;
static int _REDGLOW = REDGLOW;
static int _GREENPLASTIC = GREENPLASTIC;
static int _BLUEPLASTIC = BLUEPLASTIC;
static int _RED = RED;
static int _WHITE = WHITE;
static int _BLACK = BLACK;

//-------------------------------------------------------------------
// end of file material support.h
//-------------------------------------------------------------------
```

```
//-----------------------------------------------------------------------
//FILENAME:  materialsupport.C
//AUTHOR: Doreen M. Jones (derived from M. Zyda Class notes
//DATE: 31 MAY 97
//COMPILER: SUN/UNIX on SGI
//SUMMARY: These functions control the lighting and the colors used
// in drawing the scene.
//-----------------------------------------------------------------------

#include <Xm/Xm.h>            // Get the Motif stuff...
#include <GL/GLwMDrawA.h>     // We are going to use an OpenGL Motif Draw widget
#include <GL/gl.h>            // Get the OpenGL required includes.
#include <GL/glu.h>
#include <GL/glx.h>

#include <iostream.h>         // C++ I/O subsystem.
#include <math.h>
#include <stdlib.h>           // Get exit() function.

#include "materialsupport.h"  // Get the names of the available materials.
#include "materialsupport_funcs.h"  // Get material support function names.
#include "drawsupport_funcs.h"


//-----------------------------------------------------------------------
//function: turnOnTheLightModel
//description: The following function turns on the Light Model.
//-----------------------------------------------------------------------

void turnOnTheLightModel()
{

   // A dim white for the lighting model ambient color.
   GLfloat lmodel_ambient[] = { 0.5, 0.5, 0.5, 1.0 };

   // Set the global ambient light intensity.
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);

   // Set whether the viewpoint position is local to the scene or
   // whether it should be considered to be an infinite distance away.
   // We have selected an infinite viewpoint.
   glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

   // Say that we want two-sided lighting.
   glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

   // Enable the lighting model (there is a glDisable()).
   glEnable(GL_LIGHTING);

}
```

```
//-----------------------------------------------------------------------
//function: turnOffTheLightModel
//description: turns off the light model
//-----------------------------------------------------------------------

void turnOffTheLightModel()
{
   glDisable(GL_LIGHTING);
}


//-----------------------------------------------------------------------
//function: turnOnTheLights
//description: creates the specific lights within the scene.  Only
// one light has been added to the scene to speed up the program on
// slower machines.  This light was originally modified to allow for
// spotlights.  The functionality has been retained but is not used
// in show_war.C graphics program.
//-----------------------------------------------------------------------

void turnOnTheLights(int type)
{
   int locate;

   if (type == 0) {
     locate = 750;
   }
   else {
     locate = 350;
   }

   int place = 511;

   glTranslatef(0, 0, -511.0);
      GLfloat light0_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
      GLfloat light0_diffuse[] = {  1.0, 1.0, 1.0, 1.0 };
      GLfloat light0_specular[] = {  1.0, 1.0, 1.0, 1.0 };
      GLfloat light0_position[] = {1.0, 1.0, 1.0, 0.0 };
      glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
      glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
      glLightfv(GL_LIGHT0, GL_SPECULAR, light0_specular);
      glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
      // If we had a local light, we could have attenuation ...
      glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1.0);
      glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.0);
      glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.0);
      glEnable(GL_LIGHT0);
   glTranslatef(0, 0, place);
}


//-----------------------------------------------------------------------
//function: turnOnTheLights
//description: just turns off the light after leaving the room
//-----------------------------------------------------------------------

void turnOffTheLights()
{
   glDisable(GL_LIGHT0);
}
```

```
//-----------------------------------------------------------------
//function: turnOnMateria
//description: this creates the colors used within the simulation and
// includes some extra colors.  The included colors are:
//    BRASS,SHINYBRASS,PEWTER, SILVER, GOLD, SHINYGOLD, PLASTER,
//    REDGLOW, GREENPLASTIC, BLUEPLASTIC, RED, WHITE, BLACK
//-----------------------------------------------------------------

void turnOnMaterial(GLenum whichFace, int whichMaterial)
{

   // Set the appropriate material type...
   switch(whichMaterial)
   {
      case BRASS:
         {
            // Here are the brass material values ...
            GLfloat brass_ambient[] = { 0.35, 0.25, 0.1, 1.0 };
            GLfloat brass_diffuse[] = { 0.65, 0.5, 0.35, 1.0 };
            GLfloat brass_specular[] = { 0.65, 0.5, 0.35, 1.0 };
            GLfloat brass_shininess[] = { 5.0 };
            GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
            // Make the brass material calls.
            makeGLMaterialCalls(whichFace,
                                brass_ambient,
                                brass_diffuse,
                                brass_specular,
                                brass_shininess,
                        emmissiveness);
         }
         break;

      case SHINYBRASS:
         {
            // Here are the shinybrass material values ...
            GLfloat shinybrass_ambient[] = { 0.25, 0.15, 0.0, 1.0 };
            GLfloat shinybrass_diffuse[] = { 0.65, 0.5, 0.35, 1.0 };
            GLfloat shinybrass_specular[] = { 0.9, 0.6, 0.0, 1.0 };
            GLfloat shinybrass_shininess[] = { 10.0 };
            GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
            // Make the shinybrass material calls.
            makeGLMaterialCalls(whichFace,
                                shinybrass_ambient,
                                shinybrass_diffuse,
                                shinybrass_specular,
                                shinybrass_shininess,
                        emmissiveness);
         }
         break;

      case PEWTER:
         {
            // Here are the pewter material values ...
            GLfloat pewter_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
            GLfloat pewter_diffuse[] = { 0.6, 0.55, 0.65, 1.0 };
            GLfloat pewter_specular[] = { 0.9, 0.9, 0.95, 1.0 };
            GLfloat pewter_shininess[] = { 10.0 };
            GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
            // Make the pewter material calls.
            makeGLMaterialCalls(whichFace,
                                pewter_ambient,
                                pewter_diffuse,
                                pewter_specular,
                                pewter_shininess,
```

```
                    emmissiveness);
      }
      break;

   case SILVER:
      {
         // Here are the silver material values ...
         GLfloat silver_ambient[] = { 0.4, 0.4, 0.4, 1.0 };
         GLfloat silver_diffuse[] = { 0.3, 0.3, 0.3, 1.0 };
         GLfloat silver_specular[] = { 0.9, 0.9, 0.95, 1.0 };
         GLfloat silver_shininess[] = { 30.0 };
         GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
         // Make the silver material calls.
         makeGLMaterialCalls(whichFace,
                             silver_ambient,
                             silver_diffuse,
                             silver_specular,
                             silver_shininess,
                    emmissiveness);
      }
      break;

   case GOLD:
      {
         // Here are the gold material values ...
         GLfloat gold_ambient[] = { 0.4, 0.2, 0.0, 1.0 };
         GLfloat gold_diffuse[] = { 0.9, 0.5, 0.0, 1.0 };
         GLfloat gold_specular[] = { 0.7, 0.7, 0.0, 1.0 };
         GLfloat gold_shininess[] = { 10.0 };
         GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
         // Make the gold material calls.
         makeGLMaterialCalls(whichFace,
                             gold_ambient,
                             gold_diffuse,
                             gold_specular,
                             gold_shininess,
                    emmissiveness);
      }
      break;

   case SHINYGOLD:
      {
         // Here are the shinygold material values...
         GLfloat shinygold_ambient[] = { 0.4, 0.2, 0.0, 1.0 };
         GLfloat shinygold_diffuse[] = { 0.9, 0.5, 0.0, 1.0 };
         GLfloat shinygold_specular[] = { 0.9, 0.9, 0.0, 1.0 };
         GLfloat shinygold_shininess[] = { 20.0 };
         GLfloat emmissiveness[] = { 0.8, 0.0, 0.0, 1.0 };
         // Make the shinygold material calls.
         makeGLMaterialCalls(whichFace,
                             shinygold_ambient,
                             shinygold_diffuse,
                             shinygold_specular,
                             shinygold_shininess,
                    emmissiveness);
      }
      break;

   case PLASTER:
      {
         // Here are the plaster material values ...
         GLfloat plaster_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
         GLfloat plaster_diffuse[] = { 0.95, 0.95, 0.95, 1.0 };
         GLfloat plaster_specular[] = { 0.0, 0.0, 0.0, 1.0 };
```

```
      GLfloat plaster_shininess[] = { 1.0 };
      GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
      // Make the plaster material calls.
      makeGLMaterialCalls(whichFace,
                          plaster_ambient,
                          plaster_diffuse,
                          plaster_specular,
                          plaster_shininess,
                     emmissiveness);
   }
   break;

case REDGLOW:
   {
      // Here are the redplastic material values ...
      GLfloat redglow_ambient[] = { 0.0, 0.0, 0.0, 0.0 };
      GLfloat redglow_diffuse[] = { 0.0, 0.0, 0.0, 0.0 };
      GLfloat redglow_specular[] = { 0.0, 0.0, 0.0, 0.0 };
      GLfloat redglow_shininess[] = { 0.0 };
      GLfloat emmissiveness[] = { .0001, 0.0, 0.0, 0.0 };
      // Make the redplastic material calls.
      makeGLMaterialCalls(whichFace,
                          redglow_ambient,
                          redglow_diffuse,
                          redglow_specular,
                          redglow_shininess,
                    emmissiveness);
   }
   break;

case GREENPLASTIC:
   {
      // Here are the greenplastic material values ...
      GLfloat greenplastic_ambient[] = { 0.1, 0.3, 0.1, 1.0 };
      GLfloat greenplastic_diffuse[] = { 0.1, 0.5, 0.1, 1.0 };
      GLfloat greenplastic_specular[] = { 0.45, 0.45, 0.45, 1.0 };
      GLfloat greenplastic_shininess[] = { 30.0 };
      GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
      // Make the greenplastic material calls.
      makeGLMaterialCalls(whichFace,
                          greenplastic_ambient,
                          greenplastic_diffuse,
                          greenplastic_specular,
                          greenplastic_shininess,
               emmissiveness);
   }
   break;

case BLUEPLASTIC:
   {
      // Here are the blueplastic material values ...
      GLfloat blueplastic_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
      GLfloat blueplastic_diffuse[] = { 0.0, 0.0, 0.0, 1.0 };
      GLfloat blueplastic_specular[] = { 0.0, 1.0, 1.0, 1.0 };
      GLfloat blueplastic_shininess[] = { 10.0 };
      GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
      // Make the blueplastic material calls.
      makeGLMaterialCalls(whichFace,
                          blueplastic_ambient,
                          blueplastic_diffuse,
                          blueplastic_specular,
                          blueplastic_shininess,
                    emmissiveness);
   }
```

```
      break;

   case RED:
      {
         // Here are the red material values...
         GLfloat red_ambient[] = { 0.2, 0.0, 0.0, 1.0 };
         GLfloat red_diffuse[] = { 1.0, 0.0, 0.0, 1.0 };
         GLfloat red_specular[] = { 1.0, 0.0, 0.0, 1.0 };
         GLfloat red_shininess[] = { 100.0 };
         GLfloat emmissiveness[] = { 0.8, 0.0, 0.0, 1.0 };
         // Make the red material calls.
         makeGLMaterialCalls(whichFace,
                             red_ambient,
                             red_diffuse,
                             red_specular,
                             red_shininess,
                     emmissiveness);
      }
      break;

   case WHITE:
      {
         // Here are the white material values...
         GLfloat white_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
         GLfloat white_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
         GLfloat white_specular[] = { 1.0, 1.0, 1.0, 1.0 };
         GLfloat white_shininess[] = { 1.0 };
         GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
         // Make the red material calls.
         makeGLMaterialCalls(whichFace,
                             white_ambient,
                             white_diffuse,
                             white_specular,
                             white_shininess,
                     emmissiveness);
      }
      break;

   case BLACK:
      {
         // Here are the white material values...
         GLfloat black_ambient[] = { 0.0, 0.0, 0.0, 0.0 };
         GLfloat black_diffuse[] = { 0.0, 0.0, 0.0, 0.0 };
         GLfloat black_specular[] = { 0.0, 0.0, 0.0, 0.0 };
         GLfloat black_shininess[] = { 10.0 };
         GLfloat emmissiveness[] = { 0.0, 0.0, 0.0, 0.0 };
         // Make the red material calls.
         makeGLMaterialCalls(whichFace,
                             black_ambient,
                             black_diffuse,
                             black_specular,
                             black_shininess,
                     emmissiveness);
      }
      break;

   default:
      break;

}   // end switch statement.

}
```

```
//-----------------------------------------------------------------------
//function: turnOnMaterial
//description: turns on the material to the color specified in the
// function above.
//-----------------------------------------------------------------------

void makeGLMaterialCalls(GLenum whichFace,
                         GLfloat *ambient,
                         GLfloat *diffuse,
                         GLfloat *specular,
                         GLfloat *shininess,
                      GLfloat *emmissiveness)
{

   // Make the material calls.
   glMaterialfv(whichFace, GL_AMBIENT, ambient);
   glMaterialfv(whichFace, GL_DIFFUSE, diffuse);
   glMaterialfv(whichFace, GL_SPECULAR, specular);
   glMaterialfv(whichFace, GL_SHININESS, shininess);
   glMaterialfv(whichFace, GL_EMISSION, emmissiveness);

}
//-----------------------------------------------------------------------
// end of file material.C
//-----------------------------------------------------------------------
```

```
//-------------------------------------------------------------
//NAME: drawsSupport.h
//AUTHOR: Doreen M. Jones (derived from M. Zyda course notes)
//DATE: May 31, 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY:this file contains the function defs for drawsupport.C
//   and drawbot.C
//-------------------------------------------------------------

//-------------------------------------------------------------
// drawSupport.C function prototypes
//-------------------------------------------------------------

void drawSphere(float x, float y, float z, float radius, int nslices, int nstacks);
void drawCube(float x, float y, float z, float sidelength);
void drawCylinder(float x, float y, float z,
                  float baseradius, float topradius, float height,
                  int nslices, int nstacks);
void drawDisk(float x, float y, float z,
                  float outradius,
                  int nslices, int nloops);
void drawDisk2(float x, float y, float z,
                  float outradius, float inradius,
                  int nslices, int nloops);
void drawPartDisk(float x, float y, float z,
                  float outradius,
                  float start, float stop);
void drawthefloor(float x, float y, float z, float sidelength);
void drawthefloor2(float x, float y, float z, float sidelength);
void drawthefloor3(float x, float y, float z, float sidelength);

//-------------------------------------------------------------
// drawBot.C function prototypes.
//-------------------------------------------------------------

void drawRobot(int color);
void PaintBotA(int start, int end, char *name, int tracks, int &reset);
void PaintBotB(int start, int end, char *name, int tracks, int &reset);

//-------------------------------------------------------------------
// end of file for drawSupport.h
//-------------------------------------------------------------------
```

```
//-----------------------------------------------------------------
// NAME: drawsupport.C
// AUTHOR: Doreen M. Jones (derived from M. Zyda course notes)
// DATE: May 31, 1997
// COMPILER: SUN/UNIX on SGI
// SUMMARY: This file contains a series of draw routines for
// geometric primatives: box,  solid disk, open disk, partial disk,
// cylinder, sphere, and three different floors styles.
//-----------------------------------------------------------------

#include <Xm/Xm.h>            // Get the Motif stuff...

#include <GL/GLwMDrawA.h>     // We are going to use an OpenGL Motif Draw widget

#include <GL/gl.h>            // Get the OpenGL required includes.
#include <GL/glu.h>
#include <GL/glx.h>

#include <iostream.h>            // C++ I/O subsystem.
#include <math.h>
#include <stdlib.h>              // Get exit() function.

#include "drawsupport_funcs.h"   // Get the drawing function defs.
#include "materialsupport.h"     // get the colors
#include "materialsupport_funcs.h"

//-----------------------------------------------------------------
//FUNCTION:drawSphere
//DESCRIPTION: This function draws a sphere, masking the OpenGL
// Utility functions that are needed to draw a sphere, at the
// specified location and size.
//-----------------------------------------------------------------

void drawSphere(float x, float y, float z, float radius, int nslices, int nstacks)
{

   // Allocate a new Quadric object.
   GLUquadricObj *qobj = gluNewQuadric();

   // Say we are going to fill the polygons of the sphere.
   gluQuadricDrawStyle(qobj, GLU_FILL);

   // Save the top of the ModelView stack.
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();

      // The canonical Sphere is around the origin so move it where you want it.
      glTranslatef(x, y, z);

      // Render the Sphere.
      gluSphere( qobj, (GLdouble) radius, nslices, nstacks);

   // Restore the top of the ModelView stack.
   glPopMatrix();

   // Get rid of the space occupied by the quadric object.
   // Consider moving the creation and deletion outside or
   // consider stuffing the Sphere into a display list.
   gluDeleteQuadric(qobj);

}
```

```
//-----------------------------------------------------------------
//FUNCTION: drawCube
//DESCRIPTION: Draw a cube with center (x,y,z), having the designated
//sidelength.
//-----------------------------------------------------------------

void drawCube(float x, float y, float z, float sidelength)
{

   float halfside;   // length of half the side of the cube.
   float longside;
   float p[4][3];    // array to hold coords for the cube faces.

   halfside = sidelength/2.0;   // Compute the halfside.
   longside = sidelength;

   // Back face.
   p[0][0]=x-halfside;
   p[0][1]=y+halfside;
   p[0][2]=z-longside;

   p[1][0]=x+halfside;
   p[1][1]=y+halfside;
   p[1][2]=z-longside;

   p[2][0]=x+halfside;
   p[2][1]=y-halfside;
   p[2][2]=z-longside;

   p[3][0]=x-halfside;
   p[3][1]=y-halfside;
   p[3][2]=z-longside;

   glNormal3f(0.0, 0.0, -1.0);

   glBegin(GL_QUADS);
      glVertex3fv(p[0]);
      glVertex3fv(p[1]);
      glVertex3fv(p[2]);
      glVertex3fv(p[3]);
   glEnd();

   // draw the front face.
   p[0][0]=x+halfside;
   p[0][1]=y+halfside;
   p[0][2]=z+longside;

   p[1][0]=x-halfside;
   p[1][1]=y+halfside;
   p[1][2]=z+longside;

   p[2][0]=x-halfside;
   p[2][1]=y-halfside;
   p[2][2]=z+longside;

   p[3][0]=x+halfside;
   p[3][1]=y-halfside;
   p[3][2]=z+longside;

   glNormal3f(0.0, 0.0, 1.0);

   glBegin(GL_QUADS);
      glVertex3fv(p[0]);
      glVertex3fv(p[1]);
```

```
   glVertex3fv(p[2]);
   glVertex3fv(p[3]);
glEnd();

// Draw the top.
p[0][0]=x-halfside;
p[0][1]=y+halfside;
p[0][2]=z-longside;

p[1][0]=x-halfside;
p[1][1]=y+halfside;
p[1][2]=z+longside;

p[2][0]=x+halfside;
p[2][1]=y+halfside;
p[2][2]=z+longside;

p[3][0]=x+halfside;
p[3][1]=y+halfside;
p[3][2]=z-longside;

glNormal3f(0.0, 1.0, 0.0);

glBegin(GL_QUADS);
   glVertex3fv(p[0]);
   glVertex3fv(p[1]);
   glVertex3fv(p[2]);
   glVertex3fv(p[3]);
glEnd();

// Draw the bottom.
p[0][0]=x-halfside;
p[0][1]=y-halfside;
p[0][2]=z-longside;

p[1][0]=x+halfside;
p[1][1]=y-halfside;
p[1][2]=z-longside;

p[2][0]=x+halfside;
p[2][1]=y-halfside;
p[2][2]=z+longside;

p[3][0]=x-halfside;
p[3][1]=y-halfside;
p[3][2]=z+longside;

glNormal3f(0.0, -1.0, 0.0);

glBegin(GL_QUADS);
   glVertex3fv(p[0]);
   glVertex3fv(p[1]);
   glVertex3fv(p[2]);
   glVertex3fv(p[3]);
glEnd();

// Draw the left side.
p[0][0]=x-halfside;
p[0][1]=y-halfside;
p[0][2]=z-longside;

p[1][0]=x-halfside;
p[1][1]=y-halfside;
p[1][2]=z+longside;
```

```
   p[2][0]=x-halfside;
   p[2][1]=y+halfside;
   p[2][2]=z+longside;

   p[3][0]=x-halfside;
   p[3][1]=y+halfside;
   p[3][2]=z-longside;

   glNormal3f(-1.0, 0.0, 0.0);

   glBegin(GL_QUADS);
      glVertex3fv(p[0]);
      glVertex3fv(p[1]);
      glVertex3fv(p[2]);
      glVertex3fv(p[3]);
   glEnd();

   // Draw the right side.
   p[0][0]=x+halfside;
   p[0][1]=y-halfside;
   p[0][2]=z-longside;

   p[1][0]=x+halfside;
   p[1][1]=y+halfside;
   p[1][2]=z-longside;

   p[2][0]=x+halfside;
   p[2][1]=y+halfside;
   p[2][2]=z+longside;

   p[3][0]=x+halfside;
   p[3][1]=y-halfside;
   p[3][2]=z+longside;

   glNormal3f(1.0, 0.0, 0.0);

   glBegin(GL_QUADS);
      glVertex3fv(p[0]);
      glVertex3fv(p[1]);
      glVertex3fv(p[2]);
      glVertex3fv(p[3]);
   glEnd();
   // the cube is drawn.
}



//-----------------------------------------------------------------
//FUNCTION: drawCylinder
//DESCRIPTION: This function draws a cylinder, masking the OpenGL Utility functions
// that are needed to draw a cylinder, at the specificed location
// in space, with the height, and top and bottom radii designated.
//-----------------------------------------------------------------

void drawCylinder(float x, float y, float z,
                  float baseradius, float topradius, float height,
                  int nslices, int nstacks)
{

   // Allocate a new Quadric object.
   GLUquadricObj *qobj = gluNewQuadric();

   // Say we are going to fill the polygons of the cylinder.
   gluQuadricDrawStyle(qobj, GLU_FILL);
```

```
   // Save the top of the ModelView stack.
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();

      // The canonical Cylinder is along the z axis.
      // so shove it back half of height.
      glTranslatef(0.0, 0.0, -height/2.0);

      // Now put the center of the cylinder at x, y, z.
      glTranslatef(x, y, z);
      glRotatef(90.0, 1, 0, 0);
      // Render the Cylinder.
      gluCylinder(qobj, baseradius, topradius, height, nslices, nstacks);

   // Restore the top of the ModelView stack.
   glPopMatrix();

   // Get rid of the space occupied by the quadric object.
   // Consider moving the creation and deletion outside or
   // consider stuffing the Sphere into a display list.
   gluDeleteQuadric(qobj);

}


//-----------------------------------------------------------------
//FUNCTION: drawPartDisk
//DESCRIPTION: this draws a solid partial disk at the designated point
// in space, with a a radius, outer and a start and stop angle
//-----------------------------------------------------------------

void drawPartDisk(float x, float y, float z,
                  float outer,
                  float start, float stop)
{

   GLUquadricObj *gobj = gluNewQuadric();
   gluQuadricDrawStyle (gobj, GLU_FILL);
   glMatrixMode(GL_MODELVIEW);

   glPushMatrix();
      glTranslatef(x, y, z);
      glRotatef(90.0, 1.0, 0.0, 0.0);
      glRotatef(180.0, 0.0, 1.0, 0.0);
      gluPartialDisk (gobj, 0.0, outer, 10, 10, start, stop);
   glPopMatrix();

   gluDeleteQuadric(gobj);
}


//-----------------------------------------------------------------
//FUNCTION: drawDisk
//DESCRIPTION: draws a solid disk of radius outer at the specified
// location in space
//-----------------------------------------------------------------

void drawDisk(float x, float y, float z, float outer,
                  int nslices, int nloops)
{
   GLfloat sgenparams[] = {0.02, 0.02, 0.0, 0.0};
   GLfloat tgenparams[] = {0.02, 0.0, 0.02, 0.0};

  glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
```

```
  glTexGenfv(GL_S, GL_OBJECT_PLANE, sgenparams);
  glEnable(GL_TEXTURE_GEN_S);

  glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
  glTexGenfv(GL_S, GL_OBJECT_PLANE, tgenparams);
  glEnable(GL_TEXTURE_GEN_T);

  GLUquadricObj *gobj = gluNewQuadric();

  gluQuadricDrawStyle(gobj, GLU_FILL);

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
     glTranslatef(x, y, z);
     glRotatef(90.0, 1.0, 0.0, 0.0);
     gluDisk(gobj, 0.0, outer, nslices, nloops );
  glPopMatrix();

  gluDeleteQuadric(gobj);

  glDisable(GL_TEXTURE_GEN_S);
  glDisable(GL_TEXTURE_GEN_T);

}


//-------------------------------------------------------------------
//FUNCTION:drawDisk2
//DESCRIPTION: this draws a wire disk that can have a whole in the
// middle and a specified amount of wires at the designated point
// in space.
//-------------------------------------------------------------------

void drawDisk2(float x, float y, float z,
                  float outer, float inner,
                  int nslices, int nloops)
{
   GLfloat sgenparams[] = {0.02, 0.02, 0.0, 0.0};
   GLfloat tgenparams[] = {0.02, 0.0, 0.02, 0.0};

   glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
   glTexGenfv(GL_S, GL_OBJECT_PLANE, sgenparams);
   glEnable(GL_TEXTURE_GEN_S);

   glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
   glTexGenfv(GL_S, GL_OBJECT_PLANE, tgenparams);
   glEnable(GL_TEXTURE_GEN_T);


   GLUquadricObj *gobj = gluNewQuadric();

   gluQuadricDrawStyle(gobj, GLU_LINE);

   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
      glTranslatef(x, y, z);
      glRotatef(90.0, 1.0, 0.0, 0.0);
      gluDisk(gobj, inner, outer, nslices, nloops );
   glPopMatrix();

   gluDeleteQuadric(gobj);
   glDisable(GL_TEXTURE_GEN_S);
   glDisable(GL_TEXTURE_GEN_T);
}
```

```
//-----------------------------------------------------------------
//FUNCTION: drawthefloor()
//DESCRIPTION: this draws the square playing field floor with a large
// checker board pattern with the sides equal to sidelength.
//-----------------------------------------------------------------

void drawthefloor(float x, float y, float z, float sidelength)
{
   long i,j;   // loop temps
   float tilewidth;   // width of the tiles on the floor.
   float tx,tz;       // lowerleft coord of the floor.
   short flopcolor;   // color flop for the checkerboard.
   float v[3];        // tmp vertex hold.
   int REDFLOP   = 1;      // Some constants...
   int WHITEFLOP = 2;

   // Compute the tilewidth.
   tilewidth = sidelength/8.0;

   // Compute the lowerleft coord.
   tx = x - (4.0*tilewidth);
   tz = z - (4.0*tilewidth);

   // Set the initial color for the color flop.
   flopcolor = RED;

   // Draw the checkerboard
   for(j=0; j < 8; j=j+1)
   {

      // flop the color for the next time.
      if(flopcolor == REDFLOP)
      {
         flopcolor = WHITEFLOP;
      }
      else
      {
         flopcolor = REDFLOP;
      }

      for(i=0; i < 8; i=i+1)
      {

         /* flop the color for the next time */
         if(flopcolor == REDFLOP)
         {
            turnOnMaterial(GL_FRONT, BRASS);  // Set the red color.
            flopcolor = WHITEFLOP;
         }
         else
         {
            turnOnMaterial(GL_FRONT, PEWTER); // Set the white color.
            flopcolor = REDFLOP;
         }

         // Set the normal vector for the square.
         glNormal3f(0.0, 1.0, 0.0);

         // Draw the square for the board.
         glBegin(GL_QUADS);

            v[0] = tx+(i*tilewidth); // 1
            v[1] = y;
            v[2] = tz+(j*tilewidth);
```

```
            glTexCoord2f(0.0, 0.0); glVertex3fv(v);

            v[0] = tx+(i*tilewidth);  // 2
            v[1] = y;
            v[2] = tz+((j+1)*tilewidth);
            glTexCoord2f(1.0, 0.0);glVertex3fv(v);

            v[0] = tx+((i+1)*tilewidth);  // 3
            v[1] = y;
            v[2] = tz+((j+1)*tilewidth);
            glTexCoord2f(1.0, 1.0); glVertex3fv(v);

            v[0] = tx+((i+1)*tilewidth);  // 4
            v[1] = y;
            v[2] = tz+(j*tilewidth);
            glTexCoord2f(0.0, 1.0); glVertex3fv(v);

         glEnd();

      }   // endfor i.

   }   // endfor j.

}



//-----------------------------------------------------------------
//FUNCTION:drawthefloor2
//DESCRIPTION: this draws the square playing field floor with a small
// checker board pattern with the sides equal to sidelength.
//-----------------------------------------------------------------

void drawthefloor2(float x, float y, float z, float sidelength)
{

   long i,j;   // loop temps
   int count = 64;
   float tilewidth;   // width of the tiles on the floor.
   float tx,tz;       // lowerleft coord of the floor.
   short flopcolor;   // color flop for the checkerboard.
   float v[3];        // tmp vertex hold.
   int REDFLOP   = 1;      // Some constants...
   int WHITEFLOP = 2;

   // Compute the tilewidth.
   tilewidth = sidelength/count;

   // Compute the lowerleft coord.
   tx = x - (.5*count*tilewidth);
   tz = z - (.5*count*tilewidth);

   // Set the initial color for the color flop.
   flopcolor = RED;

   // Draw the checkerboard
   for(j=0; j < count; j=j+1)
   {

      // flop the color for the next time.
      if(flopcolor == REDFLOP)
      {
         flopcolor = WHITEFLOP;
      }
```

```
      else
      {
         flopcolor = REDFLOP;
      }

      for(i=0; i < count; i=i+1)
      {

         /* flop the color for the next time */
         if(flopcolor == REDFLOP)
         {
            turnOnMaterial(GL_FRONT, PLASTER);  // Set the red color.
            flopcolor = WHITEFLOP;
         }
         else
         {
            turnOnMaterial(GL_FRONT, BRASS); // Set the white color.
            flopcolor = REDFLOP;
         }

         // Set the normal vector for the square.
         glNormal3f(0.0, 1.0, 0.0);

         // Draw the square for the board.
         glBegin(GL_QUADS);
            v[0] = tx+(i*tilewidth); // 1
            v[1] = y;
            v[2] = tz+(j*tilewidth);
            glVertex3fv(v);

            v[0] = tx+(i*tilewidth);  // 2
            v[1] = y;
            v[2] = tz+((j+1)*tilewidth);
            glVertex3fv(v);

            v[0] = tx+((i+1)*tilewidth);  // 3
            v[1] = y;
            v[2] = tz+((j+1)*tilewidth);
            glVertex3fv(v);

            v[0] = tx+((i+1)*tilewidth);  // 4
            v[1] = y;
            v[2] = tz+(j*tilewidth);
            glVertex3fv(v);

         glEnd();

      }   // endfor i.
   }   // endfor j.
}



//-----------------------------------------------------------------
//FUNCTION:
//DESCRIPTION: this draws the square playing field floor as a
// single square with the sides equal to sidelength.
//-----------------------------------------------------------------

void drawthefloor3(float x, float y, float z, float sidelength)
{
   float halfside = sidelength/2;
   float p[4][3];
```

```
   // Draw the top.
   p[0][0]=x-halfside;
   p[0][1]=y;
   p[0][2]=z-halfside;

   p[1][0]=x-halfside;
   p[1][1]=y;
   p[1][2]=z+halfside;

   p[2][0]=x+halfside;
   p[2][1]=y;
   p[2][2]=z+halfside;

   p[3][0]=x+halfside;
   p[3][1]=y;
   p[3][2]=z-halfside;

   glNormal3f(0.0, 1.0, 0.0);
   turnOnMaterial(GL_FRONT, PLASTER);
   turnOnMaterial(GL_BACK, PLASTER);
   glPushMatrix();
      glBegin(GL_QUADS);
         glVertex3fv(p[0]);
         glVertex3fv(p[1]);
         glVertex3fv(p[2]);
         glVertex3fv(p[3]);
      glEnd();
      turnOnMaterial(GL_FRONT, GREENPLASTIC);
      drawSphere(-x-halfside, y, z-halfside, 20, 10, 10);
      turnOnMaterial(GL_FRONT, BLUEPLASTIC);
      drawSphere(halfside, y, z-halfside, 20, 10, 10);
   glPopMatrix();
}

//-----------------------------------------------------------
// end of file drawsupport.C
//-----------------------------------------------------------
```

```
//-------------------------------------------------------------
//NAME: drawBot.C
//AUTHOR: Doreen M. Jones (derived from M. Zyda course notes)
//DATE: May 31, 1997
//COMPILER: SUN/UNIX on SGI
//SUMMARY:This contains the Simbot 3D graphic and the two Simbot
// movement functions.  The two functions to draw the Simbots
// could have probably been combined into one but were not to
// provide clarity.
//-------------------------------------------------------------

#include <Xm/Xm.h>            // Get the Motif stuff...
#include <GL/GLwMDrawA.h>     // We are going to use an OpenGL Motif Draw widget
#include <GL/gl.h>            // Get the OpenGL required includes.
#include <GL/glu.h>
#include <GL/glx.h>

#include <iostream.h>         // C++ I/O subsystem.
#include <fstream.h>
#include <math.h>
#include <stdlib.h>           // Get exit() function.

#include "drawsupport_funcs.h"  // Get the drawing function defs.
#include "materialsupport.h"
#include "materialsupport_funcs.h"

static int sizeA = 0;  // size of the data chunks containing Simbot info
static int sizeB = 0;
static int locationA;  // place holders in the data files
static int locationB;


//-------------------------------------------------------------------
//function: drawRobot
//description: puts together the 3D graphic primative to form the
// Simbot.  Due to the rotation of the forms, they are not all placed
// exactly at the center point.
//-------------------------------------------------------------------

void drawRobot(int color)
{
   glPushMatrix();
      turnOnMaterial(GL_FRONT, WHITE);
      drawCylinder(0.0, -150.0, -491.0, 50.0, 50.0, 40, 20, 20);
      turnOnMaterial(GL_FRONT, BLACK);
      turnOnMaterial(GL_FRONT, SILVER);
      drawCylinder(0.0, -130.0, -501.0, 50.0, 50.0, 20, 20, 20);
      turnOnMaterial(GL_FRONT, RED);
      drawCylinder(0.0, -124, -508, 50.0, 50.0, 6, 20, 20);
      turnOnMaterial(GL_BACK, REDGLOW);
      turnOnMaterial(GL_BACK, PEWTER);
      turnOnMaterial(GL_FRONT, color);
      drawCylinder(0.0, -85.0, -491.0, 50.0, 50.0, 40, 20, 20);
      turnOnMaterial(GL_BACK, color);
      drawDisk(0.0, -86.0, -511.0, 50.5, 20, 20);
      turnOnMaterial(GL_BACK, PEWTER);
      turnOnMaterial(GL_FRONT, PEWTER);
      drawCylinder(0.0, -50.0, -491.0, 5.0, 5.0, 40, 20, 20);
      turnOnMaterial(GL_FRONT, BLACK);
      turnOnMaterial(GL_BACK, BLACK);
      turnOnMaterial(GL_FRONT, PEWTER);
      glTranslatef(0.0, -100.0, 0.0);
      glRotatef(90.0, 0.0, 0.0, 1.0);
      turnOnMaterial(GL_FRONT, PEWTER);
```

```
      drawCylinder(0, 0, -461, 5.0, 5.0, 100, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//function:PaintBotA
//description: This function accesses the data files to obtain the
// coordinates and rotation to which to draw the Simbot on the
// playing field.   The function will reaccess the data files where
// it last left.   The function will return to the start of the data
// file or at the start times listed.
//---------------------------------------------------------------------

void PaintBotA(int start, int final, char *title, int pause, int &reset)
{
   float old_loc;
   float Itheta, robx, robz;
   int Icolor, Itime;
   char discard[256];

   ifstream inBot(title, ios::in);

   old_loc = locationA;
   inBot.seekg(locationA);

   inBot >> Itime >> Itheta >> Icolor >>  robx >> robz ;
   inBot.getline(discard, 250, '\n');

   if (Itime < final && pause == 0) { //get next line if not paused
      locationA = inBot.tellg();
   }

   if (Itime == 0) { // at start determine the size of the data line
      sizeA = (locationA - old_loc);
      if (start != 1 && start > 1) {
         locationA = start * sizeA;
      }
   }

   if (reset == 2) {  // if replay or reset is indicated start over
      locationA = start * sizeA;
      reset = 0;
   }

   //cout << "robotA = ";
   //cout << Itime << ' ' << Itheta << ' ' << robx << ' ' << robz << endl;

   glPushMatrix();
      glTranslatef(0, 0, -511.0);
      glTranslatef(robx, 0, robz);
      glRotatef(Itheta, 0.0, 1.0, 0.0);
      glTranslatef(0, 0, 511.0);
      drawRobot(Icolor);
   glPopMatrix();

   return;
}
```

```
//-----------------------------------------------------------------
//function:PaintBotA
//description: This function accesses the data files to obtain the
// coordinates and rotation to which to draw the Simbot on the
// playing field.   The function will reaccess the data files where
// it last left.   The function will return to the start of the data
// file or at the start times listed.
//-----------------------------------------------------------------

void PaintBotB(int start, int final, char *title, int pause, int &reset)
{

   float old_loc;
   float Itheta, robx, robz;
   int Icolor, Itime;
   char discard[256];

   ifstream inBot(title, ios::in);

   old_loc = locationB;
   inBot.seekg(locationB);

   inBot >> Itime >> Itheta >> Icolor >>  robx >> robz ;
   inBot.getline(discard, 560, '\n');
   if (Itime < final && pause == 0) {
      locationB = inBot.tellg();
   }

   if (Itime == 0) {
      sizeB = (locationB - old_loc);
      if (start != 1 && start > 1) {
         locationB = start * sizeB;
      }
   }

   if (reset == 2) {
      locationB = start * sizeB;
      reset = 0;
   }

   //cout << "robot4 = ";
   //cout << Itime << ' ' << Itheta << ' ' << robx << ' ' << robz << endl;

   glPushMatrix();
      glTranslatef(0, 0, -511.0);
      glTranslatef(robx, 0, robz);
      glRotatef(Itheta, 0.0, 1.0, 0.0);
      glTranslatef(0, 0, 511.0);
      drawRobot(Icolor);
   glPopMatrix();

   return;
}

//-----------------------------------------------------------------
// end of file drawbot.C
//-----------------------------------------------------------------
```

APPENDIX H. VIDEO

This video appendix contains short clips of animation illustrating the Simbots interacting with each other and the playing field boundaries. The first clip shows the green Simbot using the converted Dynamic C code against the gray Simbot using a simple track and shoot algorithm. The next two clips show the Simbots colliding with each other and with the playing field boundaries.

LIST OF REFERENCES

Hofler, Tom, SE 3015 Course Notes, Naval Post Graduate School, Monterey, California, January 22, 1997.

Harkins, Richard M. SE 3015 Course Notes, Naval Post Graduate School, Monterey, California, January 22, 1997.

Kinsler, Lawrence, E. et al., *Fundamental of Acoustics*, Third Edition, John Wiley and Sons, New York, NY, 1982.

McMinds, Donald L., *Mastering OSF/Motif Widgets*, Second Edition, Addison-Wesley Publishing Company, Reading, MA, 1993.

Naval Postgraduate School, Academic Year 1995, *Course Guide*, 1995, p. 307.

Neider, Jackie, et al., *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*, Addison-Wesley Publishing Company, Reading, MA, 1993.

Simpson Robert E., *Introductory Electronics for Scientists and Engineers*, Second Edition, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1987.

Zworld Engineering, *Dynamic C Application Frameworks*, revision 1.1, Davis, CA, July 1, 1995.

Zworld Engineering, *Tiny Giant, Cprogrammable Miniature Controller Technical Reference Manual*, version 1.1, Davis, CA, November 1, 1993.

Zyda, M., OpenGL Course Notes, Naval Post Graduate School, Monterey, California, December, 1996.

BIBLIOGRAPHY

Deitel, H.M. and Deitel, P.J., *C++ How to Program*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

Fowles, Grant R., Cassiday, George L., *Analytical Mechanics*, Fifth Edition, Saunders College Publishing Harcourt Bracde College Publishers, Fort Worth, Texas, 1986.

Halliday, David, et al., *Fundamentals of Physics*, Fourth Edition, John Wiley and Sons, New York, 1993.

Hecht, Eugene, *Optics*, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusettes, 1990.

INITIAL DISTRIBUTION LIST

1. Defense Techinical Information Center.....2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virgina 22060-6218

2. Dudley Knox Library.....2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Professor Anthony A. Atchley.....1
   Chairman, Department Physics
   Naval Postgraduate School (Code PH/We)
   Monterey, California 93943

4. Professor Gordon Schachar.....1
   Naval Postgraduate School (Code PH/Sq)
   Monterey, California 93943

5. Assistant Professor Don Brutzman.....1
   Naval Postgraduate School (Code UW/Br)
   Monterey, California 93943

6. RADM Rodney Rempt.....1
   National Center 2 (PEO/TAD)
   2531 Jefferson Davis Highway
   Arlington, VA 22242-5170

7. Professor James Eagle.....1
   Chair, Department of Undersea Warfare
   Naval Postgraduate School (Code UW)
   Monterey, California 93943

8. Professor Tom Hofler.....1
   Naval Postgraduate School (Code PH/Ho)
   Monterey, California 93943

9. Professor Robert McGhee.....1
   Naval Postgraduate School (Code CS/Bz)
   Monterey, California 93943

10. LCDR Richard M. Harkins.....1
Naval Postgraduate School (Code PH/Hr)
Monterey, California 93943

11. Associate Professor Don Walters.....1
Naval Postgraduate School (PH/We)
Monterey, California 93943

12. Dr. Joe Cipriano.....1
National Center 2 (PEO/TAD-SE)
2531 Jefferson Davis Highway
Arlington, VA 22242-5170

13. LT Doreen M Jones.....1
CINCUSNAVEUR London UK
PSC 802 Box 6
FPO AE 09499-0156